اصل و نسب جاوا

جاوا به زبان C++ نتیجه مستقیم زبان C وابسته است . بسیاری از خصلتهای جاوا بطور مستقیم از این دو زبان گرفته شده است . دستور زبان جاوا منتج از دستور زبان C است . بسیاری از جنبه های oop زبان جاوا از C++ بعاریت گرفته شده است . در حقیقت بسیاری از خصلتهای زبان جاوا از این دو زبان مشتق شده یا با آنها مرتبط است . علاوه بر این ، تولید جاوا بطور عمیقی متاثر از روال پالایش و تطبیقی است که طی سه دهه گذشته برای زبانهای برنامه نویسی موجود پیش آمده است . بهمین دلایل بهتر است سیر مراحل و نیروهایی که منجر به تولد جاوا شده را بررسی نماییم . هرنوع ابتکار و فکر جدید در طراحی زبانها براساس نیاز به پشت سر نهادن یک مشکل اصلی است که زبانهای قبلی از حل آن عاجز مانده اند . جاوا نیز بهمین ترتیب متولد شد .

جاوا از نظر ساختار بسیار شبیه زبان C/C++ و این به هیچ وجه تصادفی نیست c زبانی است ساخته یافته و c++ زبانی شي گرا و مهمتر از همه قسمت اعظم برنامه نویسان دنیا از C/C++ استفاده می کنند. و از سوی دیگر این حرکت به طرف جاوا را برای این قبیل افراد ساده خواهد کرد.

جاوا با دور انداختن نشانگرها (Pointers)و بر دوش کشیدن بار مدیریت حافظه برنامه نویسان C/C++ را برای همیشه از این کابوس رهایی بخشیده است جاوا همچون C/C++ به بزرگی و کوچکی حروف حساس است و برنامه نوشته شده باید دارای متد main باشد.

زمینه های پیدایش جاوا

تاریخچه زبانهای برنامه نویسی بشرح زیر است : زبان B منجر به ظهور زبان C وC زمینه پیدایش C++ شد و در نهایت زبان جاوا متولد شد . درک زبان جاوا مستلزم : درک زمینه های لازم برای ایجاد جاوا ، نیروهایی که این زبان را شکل داده اند و مشخصاتی است که این زبان از اسلاف خود به ارث برده است . نظیر سایر زبانهای برنامه نویسی موفق ، جاوا نیز عناصر بارث برده از اسلاف خود را با ایده های ابتکاری که ناشي از محیط منحصر بفرد این زبان بوده درهم آمیخته است . فصول بعدی جنبه های عملی زبان جاوا شامل دستور زبان (syntax) و کتابخانه ها (libraries)و کاربردهای جاوا را توصیف می کند . فعلا" چگونگی و علت ظهور جاوا و اهمیت آن را بررسی می کنیم . اگر چه جاوا تفکیک ناپذیری با محیط های همزمان اینترنت پیوستگی دارد ، اما بخاطر بسپارید که جاوا قبل از هر چیز یک زبان برنامه نویسی است . ابداعات و پیشرفت ها در زبانهای برنامه نویسی کامپیوتر بدو دلیل بروز می کنند : تطابق با تغییرات محیط ها و کاربردها . ایجاد پالایش و پیشرفت در هنر برنامه نویسی . همانطوریکه بعدا" مشاهده می کنید ، تولد جاوا از این دو دلیل بطور یکسان به ارث گرفته است.

جاوا هم مانند اکثر اختراعات مهم حاصل تلاش گروهی دانشمند پیشتاز است . مدیران سان به این فکر افتادند که کاری کنند که سیستم مزبور بتواند به سیستم سخت افزاری مختلف منتقل شود . برای این منظور ابتدا از کامپایلر C++ استفاده

کنند ولی به زودی نارسایی ++C در این زمینه خود را نشان داد .و مهندسان سان خیلی سریع دریافتند که برای ادامه کار باید چیزی جدید و قوی خلق کنند .
نسخه اولیه ی جاوا در سال 1991 با نام Oak توسط تیمی از برنامه نویسان شرکت سان به سرپرستی جیمز گاسلینگ طراحی شد و در سال 1992 به جاوا تغییر نام پیدا کرد و به بازار عرضه شد .

تولد زبان برنامه نویسی جدید C :

زبان C پس از تولد ، شوك بزرگی به دنیای کامپیوتر وارد کرد . این زبان بطور اساسی شیوه های تفکر و دستیابی به برنامه نویسی کامپیوتر را دگرگون ساخت . تولدC ناشی از نیاز به یك زبان ساخت یافته ، موثر و سطح بالا بعنوان جایگزینی برای کدهای اسمبلی و پیاده سازی برنامه نویسی سیستم بود . هنگامیکه یك زبان برنامه نویسی جدید متولد میشود ، مقایسه ها شروع خواهد شد . مقایسه ها براساس معیارهای زیر انجام می گیرند : ؤ راحتی کاربری در مقایسه با قدرتمندی زبان برنامه نویسی ؤ ایمنی در مقایسه با سطح کارآیی ؤ استحکام در مقایسه با توسعه پذیری قبل از ظهور زبان C برنامه نویسان با زبانهایی کار می کردند که قدرت بهینه سازی یك مجموعه خاص از خصایص را داشتند . بعنوان مثال هنگامیکه از فرترن برای نوشتن برنامه های موثر در کاربردهای علمی استفاده می کنیم ، برنامه های حاصله برای کدهای سیستم چندان مناسب نیست . زبان بیسیك با اینکه براحتی آموخته می شود ، اما قدرت زیادی نداشته و عدم ساخت یافتگی آن در برنامه های بزرگ مشکل آفرین خواهد شد . از زبان اسمبلی برای تولید برنامه های کاملا" موثر استفاده می شود ، اما آموزش و کار با این زبان بسیار مشکل است . بعلاوه اشکال زدایی کدهای اسمبلی بسیار طاقت فرساست . مشکل اصلی دیگر این بود که زبانهای اولیه برنامه نویسی نظیر بیسیك ، کوبول و فرترن براساس اصول ساخت یافته طراحی نشده بودند . این زبانها از Goto بعنوان ابزارهای اولیه کنترل برنامه استفاده می کردند . در نتیجه ، برنامه های نوشته شده با این زبانها تولید باصطلاح " کدهای اسپاگتی (spaghetti code()" می کردند منظور مجموعه ای در هم تنیده از پرشها و شاخه های شرطی است که درك یك برنامه طولانی را ناممکن می سازد . اگر چه زبانهایی نظیر پاسکال ، ساخت یافته هستند اما فاقد کارایی لازم بوده و جنبه های ضروری برای کاربرد آنها در طیف وسیعی از برنامه ها وجود ندارد . (بخصوص ویرایش پاسکال استاندارد فاقد ابزارهای کافی برای استفاده در سطح کدهای سیستم بود) . تا قبل از ابداع زبان C ، زبان دیگری قدرت نداشت تا خصلتهای متضادی که در زبانهای قبلی مشاهده میشد ، را یکجا گردآوری کند . نیاز به وجود یك چنین زبانی شدیدا" احساس میشد. در اوایل دهه 1970 میلادی ، انقلاب رایانه ای در حال شکل گیری بود و تقاضا برای انواع نرم افزارها فشار زیادی روی برنامه نویسان و تواناییهای ایشان اعمال میکرد. درمراکز آموزشی تلاش مضاعفی برای ایجاد یك زبان برنامه نویسی برتر انجام می گرفت . اما شاید از همه مهمتر تولید و عرضه انبوه سخت افزار کامپیوتری بود که بعنوان یك نیروی ثانویه روی زبانهای برنامه نویسی عمل میکرد .دیگر رایانه ها و اسرار درونی آنها پشت درهای بسته نگهداری نمی شد . برای اولین بار بود که برنامه نویسان واقعا" دسترسی نامحدودی به اسرار ماشینهای خود پیدا نمودند . این امر زمینه تجربیات آزادانه را بوجود آورد . همچنین برنامه نویسان توانستند ابزارهای مورد نیازشان را ایجاد نمایند . با ظهور زبان C ، زمینه جهش های بزرگ در زبانهای برنامه نویسی مهیا شد . زبان C نتیجه توسعه تحقیقاتی درباره یك زبان قدیمی تر بنام Bcplبود . زبانC اولین بار توسط Dennis Ritchie ابداع و روی ماشینهای DEC PDP-11 دارای سیستم عامل یونیکس اجرا شد . زبان Bcpl توسط Martin Richards توسعه یافته بود Bcpl . منجر به تولد زبان B

شد که توسط Ken thompson ابداع شد و سرانجام به زبان C منتهی شد . برای سالیان متمادی ، نسخه روایت استاندارد زبان C همانی بود که روی سیستم عامل unix عرضه و توسط Briian Kernighan و Dennis Ritchie و در کتاب "The C programming Language" توصیف شده بود . بعدا" در سال 1989 میلادی زبان C مجددا "استاندارد شد وو استاندارد ANSI برای زبان C انتخاب شد . بسیاری معتقدند که ایجاد زبان C راهگشای دوران جدیدی در زبانهای برنامه نویسی بوده است . این زبان بطور موفقیت آمیزی تناقضهای موجود در زبان های برنامه نویسی قبلی را مرتفع نمود . نتیجه فرآیند ایجاد زبان C ، یک زبان قدرتمند ، کارا و ساخت یافته بود که براحتی قابل آموزش و فراگیری بود . این زبان یک ویژگی غیر محسوس اما مهم داشت : زبان C ، زبان برنامه نویسان بود . قبل از ابداع

زبان C

زبانهای برنامه نویسی یا جنبه های آموزشی داشته یا برای کارهای اداری طراحی میشد . اما زبان C چیز دیگری بود . این زبان توسط برنامه نویسان واقعی و درگیر با کارهای جدی ، طراحی و پیاده سازی شده و توسعه یافت . جنبه های مختلف این زبان توسط افرادی که با خود زبان سر و کار داشته و برنامه نویسی می کردند مورد بررسی ، آزمایش و تفکر و تفکر مجدد قرار گرفته بود . حاصل این فرآیند هم زبانی بود که برنامه نویسان آن را دوست داشتند . در حقیقت زبان C بسرعت مورد توجه برنامه نویسان قرار گرفت تا جایی که برنامه نویسان نسبت به C تعصب خاصی پیدا نمودند . این زبان مقبولیت و محبوبیت زیادی در بین برنامه نویسان یافت . بطور خلاصه زبان C توسط برنامه نویسان و برای برنامه نویسان طراحی شده است . بعدا "می بینید که جاوا نیز این ویژگی را از اجداد خود بارث برده است .

نیاز به C++

طی دهه 1970 و اوایل دهه 80 میلادی زبان C نگین انگشتری برنامه نویسان بود و هنوز هم در سطح وسیعی مورد استفده قرار می گیرد . از آنجاییکه C یک زبان موفق و سودمند بوده ، ممکن است بپرسید چه نیازی به زبانهای جدیدتر وجود داشته است . پاسخ شما یک کلمه یعنی پیچیدگی (Complexity) است . طی تاریخ کوتاه برنامه نویسی پیچیدگی فزاینده برنامه ها نیاز برای شیوه های بهتر مدیریت پیچیدگی را بوجود آورده است C++ . پاسخی است به این نیاز مدیریت پیچیدگی برنامه ها که زمینه اصلی پیدایش C++ بوده است . شیوه های برنامه نویسی از زمان اختراع رایانه تاکنون بطور قابل توجهی تغییر نموده اند . بعنوان مثال ، هنگامیکه رایانه ها اختراع شدند ، برنامه نویسی با استفاده از دستور العملهای باینری (Binary) ماشین انجام می گرفت . مادامیکه برنامه ها شامل حدود چند دستور العمل بود ، این روش کارآیی داشت . بموازات رشد برنامه ها زبان اسمبلی ابداع شد تا برنامه نویسان بتوانند برنامه های بزرگتر و پیچیده تر را با استفاده از نشانه هایی که معرف دستورالعملهای ماشین بودند ، بنویسند . اما پیشرفت و رشد برنامه ها همچنان ادامه یافت و زبانهای سطح بالایی معرفی شدند که ابزارهای مناسب برای مدیریت پیچیدگی روزافزون برنامه ها را در اختیار برنامه نویسان قرار می دادند .

اولین زبان مطرح در این زمینه فرترن بود . اگر چه فرترن اولین گام در این مسیر بود، اما زبانی است که توسط آن برنامه

های تمیز و سهل الادراک نوشته میشود .در دهه 1960 میلادی برنامه نویسی ساخت یافته مطرح شد . با استفاده از زبانهای ساخت یافته ، برای اولین بار امکان راحت نوشتن برنامه های بسیار پیچیده بوجود آمد . اما حتی با وجود روشهای برنامه نویسی ساخت یافته ، هنگامیکه یک پروژه به اندازه معینی می رسید ، پیچیدگی آن از توان مدیریت برنامه نویس خارج می شد . در اوائل دهه 1980 میلادی بسیاری از پروژه های مدیریت برنامه نویسی از مرزهای برنامه نویسی ساخت یافته گذشتند . برای حل این قبیل مشکلات ، یک روش نوین برنامه نویسی ابداع شد . این روش را برنامه نویسی شی ئ گرا یا باختصار oop می نامند . oop با جزئیات بیشتری بعدا" در همین کتاب بررسی خواهد شد ، اما توصیف مختصر این روش عبارت است از oop : یک نوع روش شناسی برنامه نویسی است که امکان سازماندهی برنامه های پیچیده از طریق بهره گیری از سه روش : وراثت ، کپسول سازی و چند شکلی ، را ایجاد می کند . در تحلیل نهایی ، اگر چه C بزرگترین و مهمترین زبان برنامه نویسی جهان است

اما محدودیتهایی در مدیریت پیچیدگی برنامه ها دارد . هنگامیکه یک برنامه از محدوده 25000 تا 100000 خط از کدها تجاوز نماید، آنچنان پیچیده می شود که درک آن بعنوان یک برنامه کلی نـاممکن خواهد شد . C++ . این محدودیت را از بین برده و به برنامه نویس کمک می کند تا برنامه هایی از این بزرگتر را نیز درک و مدیریت نماید. C++ . در سال 1979 میلادی توسط Bjarne stoustrup هنگامیکه در آزمایشگاه بل در Marry Hill ایالت New jersy مشغول کار بود ، ابداع شد . او در ابتدا این زبان جدید را (C with classes) نامید . اما در سال 1983 میلادی نام این زبان جدید به C++ تغییر یافت . C++ تداوم زبان C بود که جنبه های oop نیز به آن اضافه می شد .از آنجایی که زبان C++ براساس زبان C شکل گرفته ، در بر گیرنده کلیه جنبه ها خسلتها (attributes)و مزایای زبان C می باشد . این عوامل دلایل قاطعی برای موفقیت حتمی C++ بعنوان یک زبان برنامه نویسی هستند . ابداع C++ در حقیقت تلاشی برای ایجاد یک زبان کاملا" جدید برنامه نویسی نبود. در حقیقت پروژه C++ منجر به افزایش تواناییهای زبان موفق C شد . چون C++ از ابتدای تولد تاکنون دارای روایتهای گوناگونی شده است ، در حال حاضر در تلاش استاندارد نمودن این زبان هستند . (اولین روایت پیشنهادی ANSI برای استاندارد C++ در سال 1994 میلادی مطرح شد .)

برنامه نویسی شی ئ گرا object-oriented programming

برنامه نویسی شی ئ گرا هسته اصلی جاوا است . در حقیقت کلیه برنامه های جاوا شی ئ گرا هستند . بر خلاف C++ که در آن امکان گزینش شی ئ گرایی وجود دارد . روشهای oop آنچنان با زبان جاوا پیوستگی دارند که حتی قبل از نوشتن یک برنامه ساده جاوا نیز باید اصول oop را فرا گیرید . بهمین دلیل این فصل را با بحث جنبه های نظری oop آغاز می کنیم . می دانید که کلیه برنامه های کامپیوتری دارای دو عضو هستند : کد و داده . علاوه بر این ، یک برنامه را میتوان بطور نظری حول محور کد یا داده اش سازماندهی نمود . یعنی بعضی برنامه ها حول محور " آنچه در حال اتفاق است " (کد) نوشته شده و سایر برنامه ها حول محور " آنچه تحت تاثیر قرار گرفته است

("داده) نوشـته مـی شـوند . اینهـا دو الگـوی مختلـف سـاخت یـک برنامـه هسـتند . روش اول را مـدل پردازش گرا (process-oriented model) می نامند . در این روش یک برنامه بعنوان کدهای فعـال روی داده ها در نظر گرفت . زبانهای رویه ای (procedural) نظیر C از این مدل بنحو موفقیت آمیزی استفاده می کننـد . امـا همانطوریکه در قسمتهای قبل عنوان شد ، بموازات رشد و گسترش برنامه ها ، این روش منجر به بروز مشکلات بیشتر و پیچیده تری خواهد شد . برای مدیریت پیچیدگی فزاینده ، دومین روش معروف به برنامه نویسی شی ئ گرا پیشـنهاد شده است . برنامه نویسی شی ئ گرا یک برنامه را حول محور داده های آن یعنـی اشـیائ و یـک مجموعـه از رابطهـا (interfaces)خوش تعریف برای آن داده ها سازماندهی می کند . یک برنامه شی ئ گرا را مـی تـوان بعنـوان داده های کنترل کننده دسترسی به کدها (data controlling access to code) تلقی نمود . بعدا" خواهیـد دیـد با شروع بکار واحد داده های کنترل کننده بسیاری ازمزایای این نوع سازماندهی نصیب شما خواهد شد .

چرا نام جاوا ؟

در سال 1991 میلادی در شرکت Sun Micro Systemsمتولد شد. این پـروژه در ابتـدا پـروژه سـبز نـام داشـت. سرپرستی پروژه را James Goslingبه عهده داشت. نتیجه کار بر این پروژه زبان oakبـود کـه در سـال 92 ایجـاد شد oak.به معنای بلوط است و زمانی که جیمز از پنجره اتاق کارش به یک درخت بلـوط نگـاه مـی کـرد، ایـن نـام را برگزید؛ اما پس از مدتی شرکت Sunتصمیم گرفت نامی بهتر برای محصول خود برگزینـد. بنـابراین افـراد تـیم پـروژه سبز به یک کافی شاپ نزدیک شرکت رفتند، تا نامی دیگر برای ایـن زبـان انتخـاب کننـد. پـس از نصـف روز بحـث و بررسی JAVA، که مخفـف نامهـای James Gosling، Arthur Van hoff و Andy bechtolsheim است به عنوان نام این زبان انتخاب شد. از آنجا که مراسم نامگذاری در کافی شاپ برگزار شده بـود، یـک فنجـان قهـوه داغ به عنوان نماد جاوا در نظر گرفته شد.

تجرید Abstraction

تجرید یک عنصر ضروری در برنامه نویسی شی ئ گرا است . افراد پیچیدگی ها را با استفاده از تجرید مدیریت می نمایند . بعنوان نمونه ، مردم درباره اتومبیل هرگز بعنوان مجموعه ای از هزاران قطعات منفک از هم تفکر نمیکنند. آنها اتومبیل را بعنوان یک شی ئ خوب تعریف شده دارای نوعی رفتار منحصر بفرد تلقی می کنند . این تجرید به مردم امکان می دهد تا از یک اتومبیل استفاده نموده و به خواربار فروشی بروند ، بدون اینکه نگران پیچیدگی اجزایی باشند که یک اتومبیل را تشکیل می دهند .

آنها قادرند براحتی جزئیات مربوط به نحوه کار موتور ، سیستم های انتقال و ترمز را نادیده بگیرند . در عوض آنها مختارند تا از اتومبیل بعنوان یک شی ئ کلی استفاده نمایند .

یکی از شیوه های قدرتمند مدیریت تجرید با استفاده از طبقه بندیهای سلسله مراتبی (hierarchical) انجام می گیرد . این امر به شما امکان می دهد تا معنی و مفهوم سیستم های پیچیده را کنار گذاشته و آنها را به اجزائ کوچک قابل مدیریت تقسیم نمایید. از دید بیرونی ، یک اتومبیل یک شی ئ منفرد است . از دید داخلی اتومبیل شامل چندین زیر سیستم است : فرمان ، ترمزها ، سیستم صوتی ، کمربندهای ایمنی ، سیستم حرارتی ، تلفن سلولی و غیره . هر یک از این زیر سیستم ها بنوبه خود از واحدهای تخصصی کوچکتری تشکیل شده اند. بعنوان نمونه ، سیستم صوتی اتومبیل شامل یک رادیو، یک پخش CD و یا یک پخش صوت است . نکته مهم این است که شما بدین ترتیب بر پیچیدگی اتومبیل (یا هر سیستم پیچیده دیگر) با استفاده از تجرید سلسله مراتبی ، فائق می آیید . تجریدهای سلسله مراتبی سیستم های پیچیده را می توان در مورد برنامه های کامپیوتری نیز پیاده سازی نمود. داده های یک برنامه پردازش گرای سنتی را می توان توسط تجرید اشیائ عضو آن ، منتقل نمود . یک ترتیب از مراحل پردازش را می توان به مجموعه ای از پیامها بین اشیائ تبدیل نمود. بدین ترتیب ، هر یک از این اشیائ رفتار منحصر بفرد خودش را تعریف خواهد کرد . می توانید با این اشیائ بعنوان موجودیتهای واقعی رفتار کنید که به پیامهایی که به آنها می گویند چکاری انجام دهند ، مرتبط و وابسته هستند . این هسته اصلی برنامه نویسی شی ئ گراست . مفاهیم شی ئ گرایی در قلب جاوا قرار گرفته اند ، همچنانکه پایه اصلی ادراکات بشری نیز هستند. مهم اینست که بفهمید این مفاهیم چگونه در برنامه های کامپیوتری پیاده سازی می شوند . خواهید دید که برنامه نویسی شی ئ گرا یک نمونه قدرتمند و طبیعی برای تولید برنامه هایی است که بر تغییرات غیر منتظره فائق آمده و چرخه حیات هر یک از پروژه های نرم افزاری اصلی

(شامل مفهوم سازی ، رشد و سالخوردگی) را همراهی می کنند . بعنوان نمونه ، هر گاه اشیائ خوش تعریف و رابطهای تمیز و قابل اطمینان به این اشیائ را در اختیار داشته باشید ، آنگاه بطور دلپذیری میتوانید قسمتهای مختلف یک سیستم قدیمی تر را بدون ترس جابجا نموده یا بکلی از سیستم خارج نمایید .

سه اصل oop

کلیه زبانهای برنامه نویسی شی ئگرا مکانیسمهایی را در اختیار شما قرار میدهند تا مدل شی ئگرا را پیاده سازی نمایید. این مدل شامل کپسول سازی (encapsulation) وراثت (inheritance) و چند شکلی (polymorphism)می باشد . اکنون نگاه دقیقتری به این مفاهیم خواهیم داشت .

کپسول سازی encapsulation

کپسول سازی مکانیسمی است که یک کد و داده مربوط با آن کد را یکجا گرد آوری نموده (در یک کپسول فرضی قرار داده) و کپسول بدست آمده را در مقابل دخالت یا سوئ استفاده های غیر مجاز محافظت می نماید . می توان کپسول سازی را بعنوان یک لفافه (wrapper) درنظر گرفت که کد داده مربوطه را نسبت به دستیابیهای غیرمعمول و غیر منتظره سایر کدهای تعریف شده در خارج از لفافه محافظت می کند . دستیابی به کد و داده موجود داخل لفافه از طریق رابطهای خوب تعریف شده کنترل خواهد شد .در دنیای واقعی ، سیستم انتقال اتوماتیک در یک اتومبیل را درنظر بگیرید . این سیستم صدها بیت از اطلاعات درباره موتور اتومبیل شما را کپسول سازی می کند : مثل سرعت حرکت شما ، شیب سطح در حال حرکت و موقعیت اهرم انتقال . شما بعنوان یک کاربر فقط یک راه برای تاثیر نهادن در این کپسول سازی پیچیده خواهید داشت : بوسیله تغییر اهرم انتقال دهنده ، اما با استفاده از سیگنالهای برگشتی یا برف پاک کن شیشه جلو ، نمی توانید روی سیستم انتقال قدرت اتومبیل تاثیری بگذارید .

بنابراین دست دنده (اهرم انتقال دنده) یک رابط خوب تعریف شده والبته منحصر بفرد برای سیستم انتقال است . مضاف بر اینکه آنچه درون سیستم انتقال اتفاق می افتد ، تاثیری بر اشیائ خارج از سیستم نخواهد داشت . بعنوان مثال ، دنده های انتقال ، چراغهای جلو اتومبیل را روشن نمی کنند . از آنجاییکه سیستم انتقال اتوموبیلها کپسول سازی شده ،

دهها تولید کننده اتومبیل قادرند سیستم های انتقال دلخواه خود را طراحی و پیاده سازی نمایند . اما از نقطه نظر کاربر اتومبیل همه این سیستم ها یکسان کار می کنند. درست همین ایده را می توان در برنامه نویسی کامپیوتر نیز پیاده سازی نمود . قدرت کدهای کپسول شده در این است که هر کسی می داند چگونه به آنها دسترسی یافته و میتواند صرفنظر از جزئیات اجرا و بدون ترس از تاثیرات جانبی از آنها استفاده نماید .

در جاوا کپسول سازی بر اساس کلاس (class) انجام می گیرد . اگر چه کلاس با جزئیات بیشتری در این کتاب بررسی خواهد شد ، اما بحث مختصر بعدی در این مورد سودمند خواهد بود .کلاس توصیف کننده ساختار و رفتاری (داده و کد) است که توسط یک مجموعه از اشیائ اشاعه خواهد یافت . هر شی ئ در یک کلاس شامل ساختار و رفتار تعریف شده توسط همان کلاس است . بهمین دلیل ، به اشیائ گاهی " نمونه هایی از یک کلاس " نیز می گویند . بنابراین ، یک کلاس یک ساختار منطقی است ، یک شی ئ دارای واقعیت فیزیکی است . وقتی یک کلاس بوجود می آورید ، در حقیقت کد و داده ای که آن کلاس را تشکیل می دهند ، مشخص می نمایید . این عناصر را اعضائ (members) یک کلاس می نامند .

بطورمشخص ، داده تعریف شده توسط کلاس را بعنوان متغیرهای عضو (member variables)یا متغیرهای نمونه (instance variables)می نامند . کدی که روی آن داده ها عمل می کند را روشهای عضو member methodsیا فقط روشها (methods) می نامند . اگر باC و++C و آشنا باشید می دانید که روش در برنامه نویسی جاوا همان تابع (function) در زبانهایC و++C و می باشد . در برنامه های خوب نوشته شده جاوا روشها توصیف کننده چگونگی استفاده از متغیرهای عضو هستند . یعنی که رفتار و رابط یک کلاس توسط روشهایی تعریف می شوند که روی داده های نمونه مربوطه عمل می کنند .

چون هدف یک کلاس پیاده سازی کپسول سازی برای موارد پیچیده است ، روشهایی برای پنهان کردن پیچیدگی اجزائ در داخل یک کلاس وجود دارد. هر روش یا متغیر داخل یک کلاس ممکن است خصوصی privateیاعمومی public باشد . رابط عمومی یک کلاس ، معرفی کننده هر چیزی است که کاربران خارج از کلاس نیاز به دانستن آنها دارند . روشها و داده های خصوصی فقط توسط کدهای عضو یک کلاس قابل دسترسی

هستند . بنابراین هر کد دیگری که عضو یک کلاس نباشد، نمی تواند به یک روش خصوصی دسترسی داشته باشد . چون اعضائ خصوصی یک کلاس ممکن است فقط توسط سایر بخشهای برنامه شما از طریق روشهای عمومی کلاس قابل دسترسی باشند، می توانید مطمئن باشید که فعل و انفعالات غیر مناسب اتفاق نخواهد افتاد . البته ، این بدان معنی است که رابط عمومی باید با دقت طراحی شود تا کارکرد داخلی یک کلاس را چندان زیاد برملا نکند .

وراثت inheritance

وراثت رویه ای است که طی آن یک شی ئ ویژگیهای شی ئ دیگری را کسب می کند . این موضوع بسیار اهمیت دارد زیرا از مفهوم طبقه بندی سلسله مراتبی حمایت می کند . همانطوریکه قبلا" گفتیم ، بسیاری از دانشها توسط طبقه بندی سلسله مراتبی قابل فهم و مدیریت میشوند .بعنوان مثال سگهای شکاری طلایی یکی از انواع طبقه بندیهای سگها هستند که بنوبه خود جزئی از کلاس پستانداران خونگرم بوده که در کلاس بزرگتری تحت عنوان حیوانات قرار می گیرند . بدون استفاده از سلسله مراتب ، باید خصوصیات هر یک از اشیائ را جداگانه توصیف نمود . اما هنگام استفاده از سلسله مراتب ، برای توصیف یک شی ئ کافی است کیفیتهایی که آن شی ئ را در کلاس مربوطه منحصر بفرد و متمایز می سازد ، مشخص نمایید . آن شی ئ ممکن است خصوصیات عمومی را از والدین خود بارث برده باشد .

بدین ترتیب در مکانیسم وراثت ، یک شی ئ می تواند یک نمونه مشخص از یک حالت عمومی تر باشد . اجازه دهید تا با دقت بیشتری به این رویه نگاه کنیم . بسیاری از افراد، دنیا را بطور طبیعی بعنوان مجموعه ای از اشیائ می دانند که در یک روش سلسله مراتبی بیکدیگر مرتبط شده اند . نظیر حیوانات ، پستانداران و سگها . اگر بخواهید حیوانات را با یک روش تجریدی توصیف نمایید ، باید برخی خصوصیات نظیر اندازه ، هوش و نوع اسکلت آنها را مشخص نمایید . حیوانات همچنین ویژگیهای خاص رفتاری دارند ، آنها تغذیه نموده ، تنفس کرده و می خوابند . این توصیف از خصوصیات و رفتار را توصیف " کلاس " حیوانات می نامند . اگر بخواهید توصیف یک کلاس مشخصتر از حیوانات مثل پستانداران داشته باشید .

بایـد خصوصـیات دقیقتـری نظیـر نـوع دنـدانها و آلات پستانداری را مشـخص نماییـد . ایـن را یـک زیـر کـلاس (subclass)از حیوانات و خود حیوانات را کلاس بالای (super class) پسـتانداران گوینـد . چـون پسـتانداران نوعی از حیوانات هستند ، بنابراین کلیه خصوصیات حیوانات را بارث برده اند. یک زیـر کـلاس ارث برنـده درحقیقـت کلیه خصوصیات اجداد خود در سلسله مراتب کلاس را به ارث می برد .

وراثت و کپسول سازی ارتباط دو جانبه و فعل و انفعالی دارند . اگر یک کلاس برخی از خصوصیات را کپسول سـازی کند ، آنگاه هر یک از زیر کلاسها همان خصوصیات بعلاوه برخی ویژگیهای خاص خـود را خواهنـد داشـت . همـین مفهوم ساده و کلیدی است که به برنامه نویسی شی ئ گرا امکان داده تا در پیچیـدگیها بجـای روش هندسـی ، بـروش خطی توسعه یابد . یک زیر کلاس جدید کلیه خصوصیات از کلیه اجداد خود را بارث مـی بـرد . ایـن امـر باعـث شـده تافعل و انفعالات غیر قابل پیش بینی صادره از کدهای دیگر موجود در سیستم وجود نداشته باشد .

چند شکلی polymorphism

چند شکلی مفهومی است که بوسیله آن یک رابط (interface) را می توان بـرای یـک کـلاس عمـومی از فعالیتهـا بکار برد . فعالیت مشخص توسط طبیعت دقیق یک حالت تعریف می شود . یک پشته را در نظر بگیرید (که در آن هر چیزی که آخر آمده ، ابتدا خارج می شود) . ممکن است برنامه ای داشته باشید که مستلزم سه نوع پشته باشـد . یـک پشته برای مقادیر عدد صحیح ، یکی برای مقادیر اعشاری و یکی هم برای کاراکترها لازم داریـد. الگـوریتمی کـه ایـن پشته ها را اجرا میکند، یکسان است ، اگرچه داده های ذخیره شده در هر یک از این پشته هـا متفـاوت خواهـد بـود. در یک زبان شی ئ گرا ، شما بایـد سـه مجموعـه متفـاوت از روالهـای (routines) پشـته ایجـاد نماییـد ، و بـرای هـر مجموعه اسامی متفاوت اختیار کنید .اما براساس مفهوم چند شکلی در جـاوا میتوانیـد یـک مجموعـه کلـی از روالهـای پشته را مشخص نمایید که همگی از یک نام استفاده کنند .در حالت کلی مفهوم چند شـکلی را مـی تـوان بـا عبـارت " یک رابط و چندین روش " توصیف نمود . بدین ترتیب قادر هستید یک رابط ژنریک (generic) را برای گروهـی از فعالیتهای بهم مرتبط طراحی نمایید . با طراحی یک رابط برای مشخص نمودن یک کـلاس عمـومی از فعالیتهـا، مـی توان پیچیدگی برنامه ها را کاهش داد. این وظیفه کامپایلر است تا عمل مشخصی (منظور همـان روش اسـت) را بـرای هر یک از حالات مختلف انتخاب نماید . شما بعنوان یک برنامه نویس لازم نیست این انتخاب را بصورت دستی انجام دهید . شما فقط کافی است رابط کلی را بیاد سپرده و از آن به بهترین وجه ممکن استفاده نمایید .

در مثال مربوط به سگها ، حس بویایی سگ نوعی چند شکلی است . اگر سگ بوی یک گربه را استشمام کند ، پـارس کرده و بدنبال گربه خواهد دوید . اگر سگ بوی غذا را استشمام کند ، بزاق دهانش ترشح کـرده و بطـرف ظـرف غـذا حرکت خواهد کرد . در هر دو حالت این حس بویایی سگ است که فعالیت می کند . تفاوت در آن چیزی است کـه

استشمام می شود ، یعنی نوع داده ای که به سیستم بویایی سگ وارد می شود . همین مفهوم کلی را می توان در جاوا پیاده سازی نمود و روشهای متفاوت درون برنامه های جاوا را ساماندهی کرد .

چند شکلی ، کپسول سازی و وراثت در تقابل با یکدیگر کار می کنند هنگامیکه مفاهیم چند شکلی ، کپسولسازی و وراثت را بطور موثری تلفیق نموده و برای تولید یک محیط برنامه نویسی بکار بریم ، آنگاه برنامه هایی تنومند و غیر قابل قیاس نسبت به مدلهای رویه گرا خواهیم داشت . یک سلسله مراتب خوب طراحی شده ازکلاسها، پایه ای است برای استفاده مکرر از کدهایی که برای توسعه و آزمایش آنها وقت و تلاش زیادی صرف نموده اید. کپسول سازی به شما امکان می دهد تا کدهایی را که به رابط عمومی برای کلاسهای شما بستگی دارند ، بدون شکسته شدن برای پیاده سازیهای دیگر استفاده نمایید . چند شکلی به شما امکان می دهد تا کدهای تمیز قابل حس ، قابل خواندن و دارای قابلیت ارتجاعی ایجاد نمایید .

از دو مثالی که تاکنون استفاده شده ، مثال مربوط به اتومبیل کاملا" قدرت طراحی شی ئ گرا را توصیف می کند . از نقطه نظر وراثت ، سگها دارای قدرت تفکر درباره رفتارها هستند ، اما اتوموبیلها شباهت بیشتری با برنامه های کامپیوتری دارند . کلیه رانندگان وسائط نقلیه با اتکائ به اصل وراثت انواع مختلفی (زیر کلاسها) از وسائط نقلیه را می رانند . خواه اتومبیل یک مینی بوس مدرسه ، یا یک مرسدس بنز ، یا یک پورشه ، یا یک استیشن خانوادگی باشد ، کمابیش یکسان عمل کرده ، همگی دارای سیستم انتقال قدرت ، ترمز ، و پدال گاز هستند . بعد از کمی تمرین با دنده های یک اتومبیل ، اکثر افراد براحتی تفاوت بین یک اتومبیل معمولی با یک اتومبیل دنده اتوماتیک را فرا می گیرند ، زیرا افراد بطور اساسی کلاس بالا ، یعنی سیستم انتقال را درک می کنند .

افرادی که با اتومبیل سر و کار دارند همواره با جوانب کپسول شده ارتباط دارند . پدالهای گاز و ترمز رابطهایی هستند که پیچیدگی سیستم های مربوطه را از دید شما پنهان نموده تا بتوانید براحتی و سهولت با این سیستم های پیچیده کار کنید . پیاده سازی یک موتور ، شیوه های مختلف ترمز و اندازه تایرهای اتومبیل تاثیری بر ارتباط گیری شما با توصیف کلاس پدالها نخواهند گذاشت.

آخرین خصوصیت ، چند شکلی ، بوسیله توانایی کارخانه های اتومبیل سازی برای اجرای طیف وسیعی از گزینه ها روی یک وسیله نقلیه منعکس می شود . بعنوان مثال کارخانه ممکن است از سیستم ترمز ضد قفل یا همان ترمزهای معمولی ، فرمان هیدرولیک یا چرخ دنده ای ، نیز از موتورهای 6،4 ،یا 8 سیلندر استفاده نماید
. در هر حال شما روی پدال ترمز فشار می دهید تا اتومبیل متوقف شود ، فرمان را می چرخانید تا جهت حرکت اتومبیل را تغییر دهید ، و برای شروع حرکت یا شتاب بخشیدن به حرکت روی پدال گاز فشار می دهید . در این موارد از یک رابط برای ایجاد کنترل روی تعداد متفاوتی از عملکردها استفاده شده است

کاملا" مشخص است که استفاده از مفاهیم کپسول سازی ، وراثت و چند شکلی باعث شده تا اجزائمنفک با یکدیگر ترکیب شده و تشکیل یک شی ئ واحد تحت عنوان اتومبیل را بدهند . همین وضعیت در برنامه های کامپیوتری مشاهده می شود . بوسیله استفاده از اصول شی ئ گرایی ، قطعات مختلف یک برنامه پیچیده را در کنار هم قرار می دهیم تا

یک برنامه منسجم ، تنومند و کلی حاصل شود در ابتدای این بخش گفتیم که کلیه برنامه نویسان جاوا خواه نـاخواه شـیئ گرا کلیه برنامه نویسان جاوا با مفاهیم کپسول سازی ، وراثت و پلی مورفیسم آشنا خواهند شد.

مروری بر جاوا

نظیر سایر زبانهای برنامه نویسی کامپیوتر ، عناصر و اجزائ جاوا مجرد یا منفک از هـم نیسـتند . ایـن اجـزائ در ارتبـاط تنگاتنگ با یکدیگر سبب بکار افتادن آن زبان می شوند . این پیوستگی اجزائ در عین حـال توصـیف یکـی از وجـوه خـاص ایـن زبـان را مشـکل مـی سـازد . غالبـا" بحـث دربـاره یکـی از جوانـب ایـن زبـان مسـتلزم داشـتن اطلاعات پیش زمینه در جوانب دیگر است . در قسمتهای بعدی به شما امکان داده برنامه های ساده ای توسط زبان جـاوا نوشته و درک نمایید. جاوا یک زبان کاملا" نوع بندی شده است

در حقیقت بخشی از امنیت و قدرتمندی جاوا ناشی از همین موضوع است . اکنون ببینیم کـه معنـای دقیـق ایـن موضـوع چیست . اول اینکه هر متغیری یک نوع دارد ، هر عبارتی یـک نـوع دارد و بـالاخره اینکـه هـر نـوع کـاملا" و دقیقـا" تعریف شده است . دوم اینکه ، کلیه انتسابها (assignments) خواه بطور مستقیم و صـریح یـا بوسـیله پارامترهـایی که در فراخوانی روشها عبور می کنند ، از نظر سازگاری انواع کنترل می شوند . بدین ترتیب اجبار خودکـار یـا تلاقـی انواع در هم پیچیده نظیر سایر زبانهای برنامه نویسی پیش نخواهد آمد. کامپایلر جاوا کلیه عبارات و پارامترهـا را کنتـرل می کند تا مطمئن شود که انواع ، قابلیت سازگاری بهم را داشته باشند . هر گونه عدم تناسب انـواع ، خطاهـایی هسـتند که باید قبل از اینکه کامپایلر عمل کامپایل نمودن کلاس را پایان دهد ، تصحیح شوند .

نکته : اگر دارای تجربیاتی در زبانهایC و C++ و هستید ، بیاد بسپارید که جاوا نسبت به هر زبـان دیگـری نـوع بنـدی شده تر است . بعنوان مثال درC و C++ و می توانید یک مقدار اعشاری را به یک عدد صحیح نسبت دهید . همچنـین در زبان C کنترل شدید انواع بین یک پارامتر و یک آرگومان انجام نمی گیرد . امـا در جـاوا ایـن کنتـرل انجـام مـی گیرد . ممکن است کنترل شدید انواع در جاوا در وهله اول کمی کسل کننده بنظر آید . اما بیاد داشته باشید که اجـرای این امر در بلند مدت سبب کاهش احتمال بروز خطا در کدهای شما

چرا جاوا برای اینترنت اهمیت دارد

اینترنت جاوا را پیشاپیش زبانهای برنامه نویسی قرار داد و در عوض جـاوا تـاثیرات پـیش برنـده ای روی اینترنـت داشـته است . دلیل این امر بسیار ساده است : جاوا سبب گسترش فضای حرکت اشیائ بطور آزادانـه در فضـای الکترونیکـی می شود . در یک شبکه ، دو نوع طبقه بندی وسیع از اشیائ در حال انتقال بـین سـرویس دهنـده و رایانـه شخصـی شـما وجـــود دارد : اطلاعـــات غیـــر فعـــال (passive) و برنامـــه هـــای فعـــال (active)و پویا . (dynamic) بعنوان نمونه هنگامیکه پست الکترونیکی e-mail خـود را مـرور مـی کنیـد ، در حال بررسی داده های غیر فعال هستید . حتی هنگامیکه یک برنامه را گرفته و بـار گـذاری مـی کنیـد ، مادامیکـه از آن برنامه استفاده نکنید کدهای برنامه بعنوان داده های غیر فعال هسـتند . امـا نـوع دوم اشـیائی کـه امکـان انتقـال بـه رایانـه

شخصی شما را دارند ، برنامه های پویا و خود اجرا هستند . چنین برنامه ای اگر چه توسط سرویس دهنده ارائه و انتقال می یابد ، اما یک عامل فعال روی رایانه سرویس گیرنده است . بعنوان نمونه سرویس دهنده قادر است برنامه ای را بوجود آورد که اطلاعات و داده های ارسالی توسط سرویس دهنده را نمایش دهد . بهمان اندازه که برنامه های پویا و شبکه ای شده موردتوجه قرار گرفته اند بهمان نسبت نیز دچار مشکلاتی در زمینه امنیت و قابلیت حمل هستند . قبل از جاوا ، فضای الکترونیکی شامل فقط نیمی از ورودیهایی بود که اکنون وجود دارند . همانطوریکه خواهید دید ، جاوادرها رابرای یک شکل جدید از برنامه ها باز نموده است :

ریز برنامه ها (applets)

ریز برنامه ها و برنامه های کاربردی جاوا از جاوا برای تولید دو نوع برنامه می توان استفاده نمود: برنامه های کاربردی (applications)و ریز برنامه ها . (applets) یک برنامه کاربردی برنامه ای است که روی رایانه شما و تحت نظارت یک سیستم عامل اجرا می شود . بدین ترتیب یک برنامه کاربردی ایجاد شده توسط جاوا مشابه برنامه های ایجاد شده توسط C و ++C و خواهد بود. هنگامیکه از جاوا برای تولید برنامه های کاربردی استفاده میکنیم ، تفاوتهای زیادی بین این زبان و سایر زبانهای برنامه نویسی مشاهده نمی کنیم . اما ویژگی جاوا برای تولید ریز برنامه ها دارای اهمیت زیادی است . یک ریز برنامه (applet) یک برنامه کاربردی است که برای انتقال و حرکت روی اینترنت و اجرا توسط یک مرورگر قابل انطباق با جاوا طراحی شده است . یک ریز برنامه در حقیقت یک برنامه ظریف جاوا است که بطور پویا در سراسر اینترنت قابل بارگذاری باشد ، درست مثل یک تصویر، یک فایل صوتی یا یک قطعه ویدئویی .تفاوت اصلی در اینست که ریزبرنامه یک برنامه کاربردی هوشمند است و شباهتی با یک تصویر متحرک یا فایل رسانه ای ندارد . بعبارت دیگر این برنامه قادر به عکس العمل در برابر ورودی کاربر و ایجاد تغییرات پویا است .

ریز برنامه های جاوا بسیار جالب و هیجان انگیزند و قادرند دو مشکل اصلی یعنی امنیت و قابلیت حمل را پشت سر بگذارند . قبل از ادامه بحث بهتر است مفهوم اصلی این دو مشکل را بیشتر مورد بررسی قرار دهیم .

امنیت

همانطوریکه خودتان هشیار هستید ، هرگاه که یک برنامه عادی (normal) را بار گذاری می کنید با خطر یک حمله ویروسی مواجه خواهید شد . قبل از جاوا اکثر کاربران ، برنامه های قابل اجرا را بتناوب گرفته و بارگذاری می کردند و قبل از اجرا برای ویروس زدایی اقدام به اسکن (Scanning) برنامه ها می کردند . با این حال بسیاری از این کاربران نسبت به حمله ویروسها به سیستم خود نگران بودند . علاوه بر ویروسها ، نوع دیگری از برنامه های مزاحم وجود دارند که باید در برابر آنها نیز ایمن ماند . این نوع برنامه ها قادرند اطلاعات خصوصی نظیر شماره کارتهای اعتباری ،ترازهای حساب بانکی و کلمات عبور برای جستجو درسیستم فایلهای محلی رایانه شما را کشف نموده و استفاده نمایند . جاوا توسط ایجاد یک دیواره آتش (firewall) بین رایانه شما و برنامه شبکه ای شده ، بر این مشکلات فائق آمده است .

هنگامیکه از یک مرورگر قابل انطباق با جاوا در وب استفاده میکنید، میتوانید با اطمینان ریزبرنامه های جاوا را بارگذاری نمایید، بدون اینکه از حمله ویروسها و برنامه های مزاحم واهمه ای داشته باشید . جاوا یک برنامه خاص جاوا را به محیط خاص اجرایی مربوطه اش منحصر کرده و اجازه دسترسی این برنامه به سایر بخشهای رایانه را نمی دهد و بدین ترتیب مشکل امنیت را حل کرده است . توانایی بارگذاری ریز برنامه ها بصورت مطمئن یکی از مهمترین جنبه های جاوا است .

قابلیت حمل

انواع مختلفی از رایانه ها و سیستم های عامل در سراسر دنیا مورد استفاده قرار می گیرند و بسیاری ازاین سیستم ها با اینترنت متصل میشوند. برای اینکه برنامه ها بتوانند بطور پویا روی انواع مختلف سیستم ها و محیط های عامل متصل به اینترنت بارگذاری شوند ، وسائلی برای تولید کدهای اجرایی و قابل حمل مورد نیاز است . همانطوریکه بزودی خواهید دید ، همان مکانیسمی که امنیت را بوجود می آورد سبب ایجاد قابلیت حمل نیز خواهد شد .

خصلتهای جاوا

هیچ بحثی درباره اصل و نسب جاوا بدون بررسی خصلتهای آن کامل نخواهد شد. اگر چه امنیت و قابلیت حمل ، نیروهای اصلی تسریع کننده روند شکل گیری جاوا بودند اما عوامل دیگری هم وجود دارند که در شکل نهایی این زبان تاثیر داشتند . این ملاحظات کلیدی توسط تیم جاوا در اصطلاحات زیر و بعنوان خصلتهای جاوا معرفی شده اند .

- ساده simple
- ایمن secure
- قابل حمل portable
- شی ئ گرا object-oriented
- قدرتمند Robust
- چند نخ کشی شده Multithreaded
- معماری خنثی Architecture-neutral
- تفسیر شده Interpreted
- عملکرد سطح بالا High-performance
- توزیع شده Distributed
- پویا Dynamic

قبلا" دو تا از این خصلتها را بررسی کرده ایم : ایمن و قابل حمل . اکنون سایر خصلتهای جاوا را یک به یک بررسی خواهیم نمود .

ساده

جاوا طوری شده که برنامه نویسان حرفه ای بسادگی آن را فراگرفته و بطور موثری بکار می برند . فرض کنیم که شما تجربیاتی در برنامه نویسی دارید ، آنگاه برای کار با جاوا مشکل زیادی نخواهید داشت . اگر قبلا" با مفاهیم شی ئ گرایی آشنا شده باشید ، آنگاه آموختن جاوا باز هم آسان تر خواهد شد . از همه بهتر اینکه اگر یک برنامه نویس مجرب C++ باشید ، حرکت بطرف جاوا بسیار راحت انجام می گیرد . چون جاوا دستور زبانC و C++ و همچنین بسیاری از جوانب شی ئ گرایی C++ را بارث برده ، اکثر برنامه نویسان برای کار با جاوا دچار مشکل نخواهند شد . علاوه بر اینکه بسیاری از مفاهیم پیچیده C++ یا در جاوا داخل نشده و یا با روشی آسان تر و ساده تر مورد استفاده قرار گرفته اند . فراسوی شباهتهای جاوا باC و C++ و خاصیت دیگری در جاوا وجود دارد که فراگیری آن را بسیار ساده تر می کند : جاوا تلاش کرده که جنبه های استثنایی و خارق العاده نداشته باشد . در جاوا ، تعداد اندکی از شیوه های کاملا" توصیف شده برای انجام یک وظیفه وجود دارد .

شی ئ گرا

اگر چه این زبان از اجداد خود تاثیر گرفته ، اما طوری طراحی نشده تا کد منبع آن قابل انطباق با سایر زبانهای برنامه نویسی باشد . این خاصیت به تیم جاوا اجازه داده تا آزادانه روی یک تخته سنگ حکاکی نمایند . یکی از نتایج این آزادی در طراحی ، یک روش تمیز، قابل استفاده و واقعیت گرا برای اشیائ (objects) است . جاوا از بسیاری از محیط های نرم افزاری اولیه براساس اشیائ مواردی را به عاریت گرفته و توازنی بین نظریه لفظ قلمی (purist) تحت عنوان " هر چیزی یک شی ء است " و نظریه واقعیت گرایی " جلوی راه من قرار نگیر " بوجود آورده است . مدل شی ئ در جاوا بسیار ساده و براحتی قابل گسترش است در حالیکه انواع ساده آن نظیر اعداد صحیح (integers) بعنوان عملکردهای سطح بالای غیر شی ئ تلقی می شوند .

قدرتمند

محیط چندگانه روی وب ایجاب کننده تقاضاهای غیر عادی برای برنامه هاست ، زیرا این برنامه ها باید در طیف وسیعی از سیستم ها اجرا شوند . بدین ترتیب در طراحی جاوا اولویت اول توانایی ایجاد برنامه های قدرتمند بوده است . برای کسب اطمینان جاوا شما را به تعداد محدودی از نواحی کلیدی محدود می کند تا مجبور شوید اشتباهات خود را در توسعه برنامه خیلی زود پیدا کنید . در همین حال جاوا شما را از نگرانی درباره بسیاری از اشتباهات رایج ناشی از خطاهای برنامه نویسی می رهاند . از آنجاییکه جاوا یک زبان کاملا" نوع بندی شده است ، هنگام کامپایل کد شما را کنترل می کند . اما این زبان کدهای شما را هنگام اجرا نیز کنترل می نماید . در حقیقت بسیاری از اشکالات هارددیسک به شیار که اغلب در حالتهای حین اجرا ایجاد می شوند ، در جاوا ناممکن شده اند . آگاهی بر اینکه آنچه

شما نوشته اید بصورتی قابل پیش بینی در شرایط متغیر عمل می کند ، یکی از جنبه های اصلی جاوا است .

برای درک بهتر قدرتمندی جاوا، دو دلیل عمده شکست برنامه ها را درنظر بگیرید :اشتباهات در مدیریت حافظه و شرایط استثنایی پیش بینی نشده (یعنی خطاهای حین اجرا) . مدیریت حافظه در محیطهای برنامه نویسی سنتی یکی از وظایف دشوار و آزار دهنده است . بعنوان نمونه درC وC++ و برنامه نویس باید بصورت دستی کل حافظه پویا را تخصیص داده و آزاد نماید . این امر گاه منجر به بروز مشکلاتی می شود . بعنوان نمونه گاهی برنامه نویسان فراموش می کنند حافظه ای راکه قبلا "تخصیص یافته ، آزاد نمایند . یا از این بدتر ، ممکن است تلاش کنند حافظه ای را که توسط بخشی از کد ایشان در حال استفاده است ، آزاد نمایند . جاوا بوسیله مدیریت تخصیص حافظه و تخصیص مجدد حافظه واقعا" این مشکلات را از میان برداشته است . (در حقیقت تخصیص مجدد کاملا" خودکار انجام می گیرد، زیرا جاوا یک مجموعه سطل آشغال برای اشیائ غیر قابل استفاده تهیه نموده است .) شرایط استثنایی در محیط های سنتی اغلب در حالتهایی نظیر " تقسیم بر صفر "یا"file not found" ا اتفاق می افتند و باید توسط ساختارهای بد ترکیب و زمخت مدیریت شوند . جاوا در این زمینه بوسیله تدارک اداره استثنائات شی ئ گرایی object-oriented این مشکل را حل کرده است . در یک برنامه خوش ساخت جاوا، کلیه خطاهای هنگام اجرا توسط برنامه شما مدیریت خواهد شد .

چند نخ کشی شده

جاوا برای تامین نیازمندیهای دنیای واقعی بمنظور ایجاد برنامه های شبکه ای و فعل و انفعالی (interactive) طراحی شده است . برای تکمیل این هدف ، جاوا از برنامه نویسی چند نخ کشی حمایت می کند که امکان نوشتن برنامه هایی به شما میدهد که در آن واحد چندین کار را انجام می دهند . سیستم حین اجرای جاوا ، یک راه حل زیبا و بسیار ماهرانه برای همزمانی چندین پردازش (process) ارائه می دهد که امکان ساخت سیستم های فعل و انفعالی که بنرمی اجرا میشوند را بوجود آورده است . راه حل سهل الاستفاده جاوا برای چند نخ کشی شده به شما امکان تفکر درباره رفتار خاص برنامه تان (و نه یک زیر سیستم از چند وظیفه ای) را می دهد .

معماری خنثی Architecture-Neutral

یکی از مشغولیتهای اساسی طراحان جاوا موضوع طول و قابلیت حمل کدها بود . یکی از مشکلات اصلی سر راه برنامه نویسان این است که تضمینی وجود ندارد تا برنامه ای را که امروز می نویسید فردا حتی روی همان ماشین اجرا شود. ارتقائ سیستم های عامل و پردازنده ها و تغییرات در منابع هسته ای سیستم ممکن است دست بدست هم داده تا یک برنامه را از کار بیندازند . طراحان جاوا تصمیمات متعدد و دشواری در جاوا و حین اجرا اتخاذ نمودند تا بتوانند این موقعیت را دگرگون نمایند . هدف آنها عبارت بود از : یکبار بنویسید ، هر جایی ، هر زمان و برای همیشه اجرا کنید . این هدف تا حد زیادی توسط طراحان جاوا تامین شد .

تفسیر شده و عملکرد سطح بالا

همانطوریکه دیدیم ، جاوا قدرت ایجاد برنامه هایی قابل انطباق بـا چنـدین محـیط را بوسیله کامپایـل کـردن یـک نـوع معرفی واسطه تحت عنوان کد بایتی پیدا کرده است . این کدها روی هر نوع سیستمی کـه یـک حـین اجـرای جـاوا را فراهم نماید ، قابـل اجـرا مـی باشـند . بسـیاری از راه حلهـای قبلـی در زمینـه برنامـه هـای چنـد محیطـه سـبب کـاهش سـطح عملکـرد برنامـه هـا شـده بـود .سـایر سیسـتم هـای تفسـیر شـده نظیـرBASIC، Tcl،وPERL و از کمبزدهـا و نارساییهای عملکرد رنج می بردند . اما جاوا طوری طراحی شده تا روی انواع cpu نیـز بخـوبی اجـرا شـود . اگـر چـه صحت دارد که جاوا تفسیر شده است اما کدهای بایتی جاوا آنچنان دقیق طراحی شده که می تـوان آنهـا را بسـادگی و بطور مستقیم به کدهای ماشین خاص شما برای عملکردهای سطح بالا ترجمه نمود . سیستم های حین اجـرای جـاوا کـه این بهینه سازی " درست در زمان " را اجرا می کنند ، سبب از دست رفتن هیچیک از مزایای کدهای مسـتقل از زیربنـا نخواهد شد. اکنون دیگر عملگرد سطح بالا و زیربناهای متعدد در تناقض با یکدیگر نیستند .

توزیع شده

جاوا مختص محیط توزیع شده اینترنت طراحی شده زیـرا پروتکـل هـای Tcp/ip را تبعیـت مـی کنـد . در حقیقـت ، دستیابی به یک منبع توسط آدرس URL چندان تفاوتی با دستیابی به یک فایل ندارد . روایت اولیه جـاوا یعنـی oak دربرگیرنده جنبه هایی برای پیام رسانی آدرسهای داخلی فضای الکترونیکی بود . این امر امکان می داد تـا اشـیائ روی دو نوع رایانه متفاوت ، پردازشهای از راه دور را اجرا نمایند . جاوا اخیرا"این رابطها را در یک بسـته نـرم افـزاری بنـام Remote Method Invocation (RMI)احیائ نموده است . این جنبه یک سطح غیر موازی از تجرد بـرای برنامه نویس سرویس گیرنده / سرویس دهنده بوجود آورده است .

پویا

همراه برنامه های جاوا، مقادیر قابل توجهی از اطلاعات نوع حین اجرا وجود دارد که برای ممیزی و حل مجدد دستیابی به اشیائ در زمان اجرا مورد استفاده قرار می گیرند . این امر باعث پیوند پویای کد در یک شیوه مطمئن و متهورانه مـی شود . این مسئله برای قدرتمندی محیط ریز برنامه (applet) کاملا" قاطع است ، جایی که اجرا بصورت پویا ارتقائ

جادوی جاوا کدهای بایتی The Byte codes

کلید اصلی جادوی جاوا برای حل مشکل امنیت و قابلیت حمل در این است که خروجی یک کامپـایلر جـاوا، کـدهای قابل اجرا نیستند، بلکه کدهای بایتی هستند . کد بایتی یک مجموعه کاملا" بهینه شده از دسـتورالعمل هـایی اسـت کـه برای اجرا توسط یک ماشین مجازی (Virtual Machine) طراحی شده که معـادل سیسـتم حـین اجـرای (run-time)جاوا باشد . یعنی سیستم حین اجرای جاوا یک مفسر (interpreter) برای کد بایتی است . این امر ممکن است تا حدی سبب شگفتی شود. همانطوریکه اطلاع دارید C++ به کد قابل اجرا کامپایل می شود . در حقیقت اکثر زبانهای برنامه نویسی مدرن طوری طراحی شده اند که قابل کامپایل نه قابل تفسیر باشند و این امر بلحاظ مسائل اجرایی است .اما این واقعیت که برنامه های جاوا قابل تفسیر شدن است به حل مشکلات پیوسـته بـا بارگـذاری برنامـه هـا روی اینترنت کمک می کند . دلیل آن روشن است .

جاوا بگونه ای طراحی شده تا یک زبان قابل تفسیر باشد. از آنجاییکه برنامه های جاوا قبل از آنکه قابل کامپایل باشند قابل تفسیر هستند، امکان استفاده از آنها در طیف گسترده ای از محیط ها وجود دارد . دلیل آن هم بسیار روشن است . فقط کافی است تا سیستم حین اجرای جاوا برای هر یک از محیط ها اجرا گردد . هنگامیکه بسته نرم افزاری حین اجرا برای یک سیستم خاص موجود باشد ، برنامه جاوا روی آن سیستم قابل اجرا خواهد شد . بیاد داشته باشید که اگر چه جزئیات سیستم حین اجرای جاوا از یک محیط تا محیط دیگر متفاوت است ، اما همه آنها یک کد بایتی جاوا را تفسیر می کنند . اگر جاوا یک زبان قابل کامپایل می بود ، آنگاه برای هر یک از انواع cpu متصل به اینترنت ، باید روایتهای مختلفی از جاوا نوشته می شد . این راه حل چندان قابل انطباق نیست. بنابراین "تفسیر" راحتترین شیوه برای ایجاد برنامه های واقعا" قابل حمل است .

این واقعیت که جاوا یک زبان قابل تفسیراست ، به مسئله امنیت هم کمک میکند .از آنجایی که اجرای هر یک برنامه های جاوا تحت کنترل سیستم حین اجرا انجام شده سیستم فوق می تواند برنامه را دربرگرفته و مانع تولید اثرات جانبی خارج از سیستم گردد . همانطوریکه خواهید دید ، مسئله امنیت نیز توسط محدودیت های خاصی که در زبان جاوا وجود دارد اعمال خواهد شد .

هنگامیکه یک برنامه تفسیر میشود، معمولا" کندتر از زمانی که به کدهای اجرایی کامپایل شود ، اجرا خواهد شد . اما در مورد جاوا این تفاوت در زمان اجرا چندان زیاد نیست .استفاده از کد بایتی امکان اجرای سریعتر برنامه هارا بوجود می آورد .یک نکته دیگر : اگر چه جاوا طوری طراحی شده تا تفسیر شود ، اما محدودیتی برای کامپایل کدهای بایتی آن به کدهای معمولی وجود ندارد . حتی اگر کامپایل پویا به کدهای بایتی اعمال شود ، همچنان جنبه های امنیتی و قابلیت حمل آن محفوظ می ماند ، زیرا سیستم حین اجرا همچنان درگیر محیط اجرایی می ماند . بسیاری از محیط های حین اجرای جاوا این روش " درست در زمان (just in time()" کامپایل نمودن کدهای بایتی به کدهای معمولی را مورد استفاده قرار می دهند . چنین عملکردی قابل رقابت با ++C می باشند .

جاوا یک زبان کاملا" نوع بندی شده است

در حقیقت بخشی از امنیت و قدرتمندی جاوا ناشی از همین موضوع است . اکنون ببینیم که معنای دقیق این موضوع چیست . اول اینکه هر متغیری یک نوع دارد ، هر عبارتی یک نوع دارد و بالاخره اینکه هر نوع کاملا" و دقیقا" تعریف شده است . دوم اینکه ، کلیه انتسابها (assignments) خواه بطور مستقیم و صریح یا بوسیله پارامترهایی که در فراخوانی روشها عبور می کنند ، از نظر سازگاری انواع کنترل می شوند . بدین ترتیب اجبار خودکار یا تلاقی انواع در هم پیچیده نظیر سایر زبانهای برنامه نویسی پیش نخواهد آمد. کامپایلر جاوا کلیه عبارات و پارامترها را کنترل می کند تا مطمئن شود که انواع ، قابلیت سازگاری بهم را داشته باشند . هر گونه عدم تناسب انواع ، خطاهایی هستند که باید قبل از اینکه کامپایلر عمل کامپایل نمودن کلاس را پایان دهد ، تصحیح شوند .

نکته : اگر دارای تجربیاتی در زبانهایC و++C و هستید ، بیاد بسپارید که جاوا نسبت به هر زبان دیگری نوع بندی شده تر است . بعنوان مثال درC و++C و می توانید یک مقدار اعشاری را به یک عدد صحیح نسبت دهید . همچنین در زبان C کنترل شدید انواع بین یک پارامتر و یک آرگومان انجام نمی گیرد .

اما در جاوا این کنترل انجام می گیرد . ممکن است کنترل شدید انواع در جاوا در وهله اول کمی کسل کننده بنظر آید . اما بیاد داشته باشید که اجرای این امر در بلند مدت سبب کاهش احتمال بروز خطا در کدهای شما می شود.

و در آخر جاوا :

- ساده
- شیء گرا
- قابل انتقال (Portable)
- توزیع‌شده (Distributed)
- کارایی بالا
- ترجمه شده (Interpreted)
- Multithreaded
- پویا
- ایمن (Secure)
- جاوا مجانی (اما Open Source نیست)

منابع :

http://www.irandevelopers.com/
http://docs.sun.com

نویسنده :

محمد باقر معموری **mamouri@ganjafzar.com**

ویراستار و نویسنده قسمت های تکمیلی :

احسان شاه بختی **zehs_sha@yahoo.com**

کتاب :

Java در 21 روز انتشارات نص
برنامه نویسی شی گرا انتشارات نص

:

انواع داده ها ، متغیرها و عملگرها در جاوا

از این قسمت به بعد سه عنصر اساسی جاوا را مورد بررسی قرار خواهیم داد : انواع داده ها ، متغیرها و آرایه ها. نظیر کلیه زبانهای جدید برنامه نویسی ، جاوا از چندین نوع داده پشتیبانی می کند . با استفاده از انواع داده ، می توانید متغیرها را اعلان نموده و آرایه ها را ایجاد کنید. خواهید دید که شیوه جاوا برای ین مسئله ، کاملا" روشن ، کارا و منسجم است .

متغیرها در جاوا

در یک برنامه جاوا ، متغیر ، اساسی ترین واحد ذخیره سازی است . یک متغیر به وسیله ترکیبی از یک شناسه ، یک نوع و یک مقدار ده اولیه اختیاری تعریف خواهد شد . علاوه بر این ، کلیه متغیرها دارای یک قلمرو هستند که رویت پذیری آنها را تعریف می کند و یک زمان حیات نیز دارند. متعاقبا" این اجزائ را مورد بررسی قرار می دهیم .

اعلان یک متغیر Declaring a variable

در جاوا کلیه متغیرها قبل از استفاده باید اعلان شوند . شکل اصلی اعلان متغیر بقرار زیر می باشد: type identifier [=value] [,identifier[=value]...];

مقدار شناسه مقدار شناسه نوع

نوع (type) یکی از انواع اتمی جاوا یا نام یک کلاس یا رابط است . (انواع کلاس و رابط بعدا" بررسی خواهد شد .) شناسه نام متغیر است . می توانید با گذاشتن یک علامت تساوی و یک مقدار ، متغیر را مقدار دهی اولیه نمایید . در ذهن بسپارید که عبارت مقدار دهی اولیه باید منتج به یک مقدار از همان نوعی(یا سازگار با آن نوع) که برای متغیر مشخص شده ، گردد . برای اعلان بیش از یک نوع مشخص شده ، از فهرست کاماهای (') جدا کننده استفاده نمایید . در زیر مثالهایی از اعلان متغیر از انواع گوناگون را مشاهده می کنید . دقت کنید که برخی از آنها شامل یک مقدار دهی اولیه هستند .

نکته

برای نوشتن توضیحات در جاوا از // یا /* */ استفاده می کنیم .

```
1. int a, b, c; // declares three int a, b, and c.
2. int d = 3, e, f = 5; // declares three more ints/ initializing // d and f.
3. byte z = 22; // initializes z.
4. double pi = 3.14159; // declares an approximation of pi.
5. char x = 'x'; // the variable x has the value 'x'.
```

شناسه هایی که انتخاب می کنید هیچ عامل ذاتی در نام خود ندارند که نوع آنها را مشخص نماید . بسیاری از خوانندگان بیاد مـی آورنـد زمانی را که FORTRAN کلیه شناسه های از I تا N را پیش تعریف نمود تا از نوع INTEGER باشند ، در حالیکه سـایر شناسـه ها از نوع REAL بودند . جاوا به هر یک از شناسه های متناسب شکل گرفته امکان اختیار هر نوع اعلان شده را داده است .

مقدار دهی اولیه پویا Dynamic initialization

اگر چه مثالهای قبلی از ثابت ها بعنوان مقدار ده اولیه استفاده کرده اند اما جاوا امکان مقداردهی اولیه بصورت پویـا را نیـز فـراهم آورده است . این موضوع با استفاده از هر عبارتی که در زمان اعلان متغیـر باشـد ، انجـام مـی گیـرد. بعنـوان مثـال ، در زیـر برنامـه کوتـاهی را مشاهده می کنید که طول ضلع یک مثلث قائم الزاویه را با داشتن طول دو ضلع مقابل محاسبه می کند :

```
class DynInit {
    public static void main(String args[] ){
        double a = 3.0, b = 4.0;
        // c is dynamically initialized
        double c = Math.sqrt(a * a + b * b);
        System.out.println("Hypotenuse is " + c);
    }
}
```

در اینجا سه متغیر محلی a ، b، c، اعلان شده اند . دو تای اولی توسط ثابت ها مقدار دهی اولیه شده اند . اما متغیر C بصـورت پویـا و بر حسب طول اضلاع مثلث قائم الزاویه (بنابر قانون فیثاغورث) مقدار دهی اولیه می شود . این برنامه از یکـی از روشـهای توکـار جـاوا یعنی sqrt ()که عضوی از کلاس Math بوده و ریشه دوم آرگومانهای خود را محاسبه میکنـد اسـتفاده کـرده اسـت . نکتـه کلیـدی اینجا است که عبارت مقدار دهی اولیه ممکن است از هر یک از اجزائ معتبر در زمان مقدار دهی اولیه ، شامل فراخوانی روشـها ، سـایر متغیرها یا الفاظ استفاده نماید .

قلمرو زمان حیات متغیرها

تابحال کلیه متغیرهای استفاده شده ، در زمان شروع روش main() اعلان می شدند .اما جاوا همچنین به متغیرها امکان می دهد تا درون یک بلوک نیز اعلام شوند . همانطوریکه قبلا" توضیح دادیم ، یک بلوک با یک ابرو باز و یک ابرو بسته محصور می شود : یک بلوک تعریف کننده یک قلمرو است . بدین ترتیب هر بار که یک بلوک جدید را شروع میکنید ، یک قلمرو جدید نیز بوجود می آورید . همانطوریکه احتمالا "از تجربیات برنامه نویسی قبلی بیاد دارید ، یک قلمرو (scope) تعیین کننده آن است که چه اشیائی برای سایر بخشهای برنامه قابل رویت هستند . این قلمرو همچنین زمان حیات (lifetime) آن اشیائ را تعیین می کند . اکثر زبانهای کامپیوتری دو طبقه بندی از قلمروها را تعریف می کنند : سراسری (global) و محلی . (local) اما این قلمروهای سنتی بخوبی با مدل موکد شی ئ گرایی جاوا مطابقت ندارند . اگر چه در جاوا هم می توان مقادیری را بعنوان قلمرو سراسری ایجاد نمود ، اما این فقط یک نوع استثنائ است و عمومیت ندارد . در جاوا قلمرو اصلی همانهایی هستند که توسط یک کلاس یا یک روش تعریف می شوند . حتی همین تمایز نیز تا حدی ساختگی و مصنوعی است . اما از آنجاییکه قلمرو کلاس دارای مشخصات و خصلتهای منحصر بفردی است که قابل استفاده در قلمرو تعریف شده توسط روش نیست ، این تمایز تا حدی محسوس خواهد بود . بخاطر تفاوتهای موجود ، بحث قلمرو کلاس (و متغیرهای اعلان شده داخل آن) این مبحث بتعوق افتاده است . در حال حاضر فقط قلمروهای تعریف شده توسط یک روش یا داخل یک روش را بررسی می کنیم . قلمرو تعریف شده توسط یک روش با یک ابروی باز شروع می شود. اما اگر آن روش دارای پارامترهایی باشد ، آنها نیز داخل قلمرو روش گنجانده خواهند شد . بعدا "نگاه دقیقتری به پارامترها خواهیم داشت و فعلا" کافی است بدانیم که پارامترها مشابه هر متغیر دیگری در یک روش کار می کنند . بعنوان یک قانون عمومی ، متغیرهای اعلان شده داخل یک قلمرو برای کدهایی که خارج از قلمرو تعریف می شوند ، قابل رویت نخواهند بود (قابل دسترسی نیستند .(بدین ترتیب ، هنگامیکه یک متغیر را درون یک قلمرو اعلان می کنید ، در حقیقت آن متغیر را محلی دانسته و آن را در مقابل دستیابیها و تغییرات غیر مجاز محافظت می کنید . در حقیقت ،قوانین قلمرو اساس کپسول سازی را فراهم می کنند . قلمروها را می توان بصورت تودرتو (nesting) محفوظ داشت . بعنوان مثال ، هر زمان یک بلوک کد ایجاد کنید ، یک قلمرو جدید تودرتو ایجاد نموده اید . هنگامیکه این واقعهروی می دهد ، قلمرو بیرونی ، قلمرو درونی را دربرمی گیرد . این بدان معنی است که اشیائ اعلان شده در قلمرو بیرونی برای کدهای داخل قلمرو درونی قابل رویت هستند اما عکس این قضیه صادق نیست . اشیائاعلان شده داخل قلمرو درونی برای بیرون قلمرو قابل رویت نخواهند بود . برای درک تاثیر قلمروهای تودرتو ، برناه ریز را در نظر بگیرید :

```
// Demonstrate block scope.
 class Scope {
 public static void main(String args[] ){

     int x; // known to all code within main

     x = 10;
    if(x == 10 ){ // start new scope
```

```
   int y = 20; // known only to this bock
    // x and y both known here.
    System.out.println("x and y :" + x + " " + y);
    x = y * 2;
 }
 // y = 100 :// Error! y not known here

 // x is still known here.
   System.out.println("x is " + x);
 }
 }
```

همانطوریکه توضیحات نشان مـی دهنـد ، متغیـر x در ابتـدای قلمـروی main() اعـلان شـده و بـرای کلیـه کـدهای متعاقب داخـل main()قابل دسترسی می باشد . داخل بلوک if متغیر y اعلان شده است . از آنجاییکه یک بلوک معرف یک قلمرو است ، yفقط برای سایر کدهای داخل بلوک خود قابل رویت است . این دلیل آن است که خارج بلوک مربوطه ، خط y=100 در خـارج توضـیح داده شده است . اگر نشانه توضیح راهنمایی را تغییر مکان دهید ، یک خطای زمـان کامپایـل (compile-time error) اتفـاق مـی افتد چون y برای بیرون از بلوک خود قابل رویت نیست . داخل بلوک if متغیر x قابل استفاده است زیرا کدهای داخـل یـک بلـوک (منظور یک قلمرو تودرتو شده است) به متغیرهای اعلان شده در یک قلمرو دربرگیرنده دسترسی دارند . داخل یـک بلـوک ، در هـر لحظه ای می توان متغیرها را اعلان نمود ، اما فقط زمانی معتبر می شوند که اعلان شده باشند . بدین ترتیب اگر یـک متغیـر را در ابتـدای یک روش اعلان می کنید، برای کلیه کدهای داخل آن روش قابل دسترس خواهد بود. بالعکس اگر یک متغیر را در انتهای یک بلوک اعلان کنید ، هیچ فایده ای ندارد چون هیچیک از کدها به آن دسترسی ندارند . بعنوان مثال این قطعه از برنامه غیر معتبر است چون نمی توان از count قبل از اعلان آن استفاده نمود :

```
// This fragment is wrong!
count = 100; // oops! cannot use count before it is declared!
int count;
```

یک نکته مهم دیگر در اینجا وجود دارد که باید بخاطر بسپارید: متغیرها زمانی ایجاد می شوند که قلمرو آن ها وارد شده باشد ، و زمانی خراب می شوند که قلمرو آنها ترک شده باشد . یعنی یک متغیر هربار که خارج از قلمـروش بـرود ، دیگـر مقـدار خـود را نگهـداری

نخواهد کرد . بنابراين ، متغيرهای اعلان شده داخل يک روش مقادير خود را بين فراخوانی های آن روش نگهداری نمی کنند . همچنين يک متغير اعلان شده داخل يک بلوک ، وقتی کـه بلـوک تـرک شـده باشـد ، مقـدار خـود را از دست خواهـد داد . بنـابراين ، زمـان حيات (lifetime)يک متغير محدود به قلمرو آن می باشد . اگر اعلان يک متغير شامل مقدار دهی اوليه آن باشد ، آنگاه هر زمان کـه به بلوک مربوطه واردشويم ، آن متغير مجددا" مقدار دهی اوليه خواهد شد . بعنوان مثال برنامه زير را در نظر بگيريد :

```
// Demonstrate lifetime of a variable.
class LifeTime {
public static void main(String args[] ){
int x;

for(x = 0; x < 3; x++ ){
int y =- 1; // y is initialized each time block is entered
System.out.println("y is :" + y); // this always prints- 1
y = 100;
System.out.println("y is now :" + y);
}
}
}
```

خروجی توليد شده توسط اين برنامه بقرار زير است :

```
y is- :1
y is now:100
y is- :1
y is now:100
y is- :1
y is now:100
```

همانطوريکه مشاهده می کنيد ، هر بار که به حلقه for داخلی وارد می شويم ، y همواره بطور مکرر مقدار اوليه 1- را اختيار ميکند . اگر چه بلافاصله به اين متغير مقدار 100 نسبت داده می شود، اما هر بار نيز مقدار خود را از دست ميدهد. و بالاخره آخرين نکته : اگر چه ميتوان بلوکها را تودرتو نمود، اما نميتوانيد متغيری را اعلان کنيد که اسم آن مشابه اسم متغيری در قلمرو بيرونی باشد. از اين نظر

جاوا با زبانهای C و C++ و متفاوت است . در زیر مثالی را مشاهده می کنید که در آن تلاش شده تا دو متغیر جدا از هم با اسم اعلان شوند . در جاوا اینکار مجاز نیست . در C و C++ و این امر مجاز بوده و دو bar کاملا" جدا خواهند ماند .

```
// This program will not compile
class ScopeErr {
     public static void main(String args[] ){
     int bar = 1;
        { // creates a new scope
           int bar = 2; // Compile-time error -- bar already defined!
        }
     }
}
```

تبدیل خودکار و تبدیل غیر خودکار انواع اگـر تجربـه قبلی برنامـه نویسی داشتـه ایـد ، پس مـی دانیـد کـه کـاملا" طبیعـی است کـه مقداری از یک نوع را به متغیری از نوع دیگر نسبت دهیم . اگـر ایـن دو نـوع سـازگار باشـند ، آنگـاه جـاوا بطـور خودکـار ایـن تبـدیل (conversion)را انجام می دهد . بعنوان مثال ، همواره امکان دارد که مقدار int را به یک متغیر long نسبت داد . اما همه انواع با یکدیگر سازگاری ندارند ، بنابراین هر گونه تبدیل انواع مجاز نخواهد بود . بعنوان نمونه ، هیچ تبدیلی از double بـه byte تعریـف نشده است . خوشبختانه ، امکان انجام تبدیلات بین انواع غیر سازگار هم وجود دارد . برای انجام اینکار ، باید از تبـدیل cast اسـتفاده کنیـد کـه امکـان یـک تبـدیل صـریح بـین انـواع غیـر سـازگار را بوجـود مـی آورد . اجـازه دهیـد تـا نگـاه دقیقتـری بـه تبـدیل خودکار انواع و تبدیل cast داشته باشیم .

تبدیل خودکار در جاوا Java's Automatic conyersions

هنگامیکه یک نوع داده به یک متغیر از نوع دیگر نسبت داده می شود ، اگر دو شرط زیر فراهم باشد ، یک تبـدیل خودکـار نـوع انجـام خواهد شد : ؤ دو نوع با یکدیگر سازگار باشند . ؤ نوع مقصد بزرگتر از نوع منبع باشد . هنگامیکه ایـن دو شـرط برقـرار باشـد ، یـک تبدیل پهن کننده (widening) اتفاق می افتد . برای مثال نوع int همواره باندازه کافی بزرگ است تا کلیه مقادیر معتبـر byte را دربر گیرد، بنابراین نیازی به دستور صریح تبدیل cast وجود ندارد. در تبدیلات پهن کننده ، انواع رقمـی شـامل انـواع عـدد صـحیح و عدد اعشاری با هر یک از انواع سازگاری دارند . اما انواع رقمی با انواع char و boolean و سازگار نیستند . همچنین انواع char و booleanو با یکدیگر سازگار نیستند . همانطوریکـه قـبلا" ذکـر شـد ، جـاوا هنگـام ذخیـره سـازی یـک ثابـت عـدد صـحیح لفظـی (Literal integer constant)به متغیرهای از انواع byte ، short،و long و ، یک تبدیل خودکار نوع را انجام می دهد .

تبدیل غیر خودکار انواع ناسازگار

اگر چه تبدیلات خودکار انواع بسیار سودمند هستند ، اما جوابگوی همه نیازها نیستند . بعنوان مثال ، ممکن است بخواهید یک مقدار intرا به یک متغیر byte نسبت دهید. این تبدیل بطور خودکار انجام نمی گیرد، زیرا یکbyte از int ز کوچکتر است .این نوع خاص از تبدیلات را گاهی تبدیل باریک کننده (narrowing conversions) می نامند ، زیرا بطور صریح مقدار را آنقدر باریک تر و کم عرض تر می کنید تا با نوع هدف سازگاری یابد . برای ایجاد یک تبدیل بین دو نوع ناسازگار ، باید از cast استفاده نمایید. cast یک تبدیل نوع کاملا" صریح است . شکل عمومی آن بقرار زیر می باشد(target - type)value :)

نوع نوع مقصد یا هدف

در اینجا نوع هدف ، همان نوعی است که مایلیم مقدار مشخص شده را به آن تبدیل کنیم . بعنوان مثال ، قطعه زیر از یک برنامه تبدیل غیر خودکار ازint به byte را اجرا می کند . اگر مقدار integer بزرگتر از دامنه یک byte باشد ، این مقدار به مدول(باقیمانده تقسیم یک integer بر دامنه byte) کاهش خواهد یافت;int a .

```
byte b;
//...
b =( byte )a;
```

هر گاه که یک مقدار اعشاری به یک عدد صحیح نسبت داده شود ، شکل دیگری از تبدیل اتفاق می افتد : بریدن ، . truncation همانطوریکه می دانید ، اعداد صحیح دارای قسمت اعشاری نیستند . بنابراین هنگامیکه یک مقدار اعشاری به یک نوع عدد صحیح نسبت داده می شود ، جزئ اعشاری از بین خواهد رفت (بریده خواهد شد) . (بعنوان مثال ، اگر مقدار 1.23 را به یک عدد صحیح نسبت دهیم ، مقدار حاصله فقط عدد 1 می باشد . مقدار 0.23 بریده (truncated) خواهد شد . البته اگر اندازه اجزائ عدد کلی آنچنان بزرگ باشد که در نوع عدد صحیح مقصد نگنجد ، آنگاه مقدار فوق به مدول دامنه نوع هدف کاهش خواهد یافت . برنامه زیر نشان دهنده برخی از تبدیلات انواع است که مستلزم تبدیل cast می باشند :

```
// Demonstrate casts.
class Conversion {
public static void main(String args[] ){
bytt b;
int i = 257;
double d = 323.142;
System.out.println("\nConversion of int to byte.");
```

```
b =( byte )i;
System.out.println("i and b " + i + " " + b);
System.out.println("\nConversion of double to int.");
i =( int )d;
System.out.println("d and i " + d + " " + i);

System.out.println("\nConversion of double to byte.");
b =( byte )d;
System.out/println("d and b " + d + " " + b);
}
}
```

خروجی این برنامه بقرار زیر می باشد :

```
Conversion of int to byte.
i and b 257 1

Conversion of double to int.
d and i 323.142 323

Conversion of double to byte.
d and b 323.142 67
```

اکنون به هر یک از این تبدیلات نگاه می کنیم . هنگامیکه مقدار 257 از طریق تبدیل cast به یک byte منتسب می شود ، نتیجه برابر باقیمانده تقسیم 257 بر 256 دامنه byte (یعنی عدد 1 است . هنگامیکه d به یک int تبدیل می شود ،) بخش خواهند رفت . هنگامیکه d به یک byte تبدیل می شود ،

نگاهی دقیقتر به متغیر

اکنون که انواع توکار را بطور رسمی توضیح داده ایم ، نگاه دقیقتری به این متغیر خواهیم داشت .

boolean	False,true		char	16.bits, one character
byte	one byte, integer		float	4bytes, single-precision.

short	bytes, integer 2		double	8bytes, double-precision.
long	.bytes, integer 8		int	4bytes, integer.
void	Return type where no return value is .required		String	N byte

متغیر عدد صحیح integer literals

احتمالا" اعداد صحیح رایجترین نوع استفاده شده در برنامه های نوع بندی شده هستند. هـر مقـدار رقمـی کلـی یـک لفـظ عـدد صـحیح است . اعداد 1، 2،، 3، و 42 مثالهای روشنی هستند . این اعداد همگی مقادیر دهدهی می باشند ، بدین معنـی کـه ایـن اعـداد در یـک مبنای ده رقمی تعریف شـده انـد . دو مبنـای دیگـر نیـز در متغیـر عـدد صـحیح قابـل اسـتفاده هسـتند : مبنـای هشـت (octal) و مبنـای 1 .()(hexadecimal)مقادیر در مبنای هشت در جاوا با یک رقم 0 پیش آیند مشخص میشوند. ارقام دهدهی معمولی نمـی تواننـد رقم 0 پیش آیند داشته باشند . بنابراین مقدار بظاهر معتبر 09 خطایی را در کامپایلر تولید می کند، زیـرا رقـم 9 خـارج از دامنـه 0تـا 7ا مبنای هشت قرار دارد .یکی دیگر از مبناهای رایج برای ارقام مورد استفاده برنامه نویسان ، مبنای 16 می باشد که با مـدول انـدازه هـای کلمه 8 تایی نظیر 8 ، 16 ، 32 و 64 بیتی کاملا "سازگاری دارد . یک ثابت درمبنای 16 را توسط OX یا ox ا مشخص مـی کنیـد . دامنـه یـک رقـم در مبنـای 16 از رقـم 0 تـا 15 و حـروفA تـا F ا (یـا a تـا f ا) بعنـوان جـایگزین ارقـام 10 تـا 15 مـی باشـد . متغیر عدد صحیح یک مقدار int تولید می کنند که در جاوا یک مقدار عدد صحیح 32بیتـی اسـت . از آنجاییکـه جـاوا شـدیدا" نـوع بندی شده است ، ممکن است تعجب کنید که چگونه می توان یک لفظ عدد صحیح را به یکی دیگر از انواع عـدد صـحیح جـاوا نظیـر byte

یا long ا نسبت داد ، بدون اینکه خطای عدم سازگاری انـواع بوجـود آیـد . خوشـبختانه چنـین حالتهـایی بسـادگی اداره مـی شـوند . هنگامیکه یک لفظ عدد صحیح به یک متغیر byte یا short ا منتسب مـی شـود، اگـر مقـدار لفـظ داخـل محـدوده نـوع هـدف قـرار داشته باشد ، خطایی تولید نخواهد شد . همچنین همواره می توان یک لفظ عدد صحیح را به یک متغیر long منتسب نمود . امـا بـرای مشخص نمودن یـک لفـظ long بایـد بطـور صـریح بـه کامپـایلر بگوییـد کـه مقـدار لفـظ از نـوع long اسـت . اینکـار را بـا الحـاق یک حرف L بزرگ یا کوچک به لفظ انجام می دهیم . بعنوان مثال ،

ox7fffffffffffffffL

9223372036854775807L

بزرگترین Long می باشد .

متغیر عدد اعشاری Floating-point literals

ارقام اعشاری معرف مقادیر دهدهی با اجزائ کسری می باشند . آنها را می توان به شکل استاندارد یا به شکل علامتگـذاری علمـی بیـان نمود . نشانه گذاری استاندارد شامل یک جزئ عدد صحیح است که بعد از آن یک نقطه و بعد از آن جزئ کسری عدد قرار می گیـرد . بعنوان مثال 2.0 یا 3.14159 یا 0.6667 معرف نشانه گذاری استاندارد معتبر در ارقام اعشاری هستند . نشانه گذاری علمی از یـک نشانه گذاری استاندارد نقطه مخصوص اعشاری بعلاوه یک پیوند که مشخص کننده توانی از عدد 10 است که باید در عدد ضرب شود استفاده می کند . توان (نما) را توسط علامت E یا e که یک رقم دهدهی بدنبال آن می آید و ممکن است مثبت یا منفی باشد ، نشـان می دهیم.

مثل 6.022 E23 یا 314159-05 E و 2e+100 .

متغیر عدد اعشاری در جاوا بصورت پیش فرض دارای دقت مضاعف (double) هستند . برای مشخص نمودن یک لفظ float بایـد یک حرف F یا f را به ثابت الحاق نمایید . همچنین میتوانید بطور صریح یک لفظ double را با الحاق یک حرف D یا d نیز انجام دهید . انجام اینکار البته اضافی است . نوع double پیش فرض 64 بیت حافظه را مصرف می کند در حالیکه نوع کم دقت تر float مستلزم 32 بیت حافظه است .

متغیر Boolean

متغیر boolean بسیار ساده هستند. یک مقدار boolean فقـط دو مقـدار منطقـی شـامل true و false و مـی توانـد داشـته باشـد . مقادیر true و false و هرگز به رقم تبدیل نمی شوند . در جاوا لفظ true مساوی یک نبوده ، همچنانکه لفـظ false معـادل صـفر نیست . در جاوا ، آنها را فقـط مـی تـوان بـه متغیرهـای اعـلان شـده بعنـوان boolean منتسـب نمـود و یـا در عبـاراتی بـا عملگرهـای booleanاستفاده نمود .

متغیر کاراکترها Character literals

کاراکترهای جاوا در مجموعه کاراکتر کدهای جهانی نمایه سازی شده اند . آنها مقادیر 16 بیتی هستند که قابل تبدیل به اعداد صـحیح بوده و با عملگرهای عدد صحیح نظیر عملگرهای اضافه و کسر نمودن اداره می شوند . یک کاراکتر لفظی همواره داخل یک علامت ' ' معرفی می شود . کلیه کاراکترهای ASCII قابل رویت می توانند بطور مستقیم به داخـل ایـن علامـت وارد شـوند ، مثـل 'a' یـا 'z' ایا '@' . برای کاراکترهایی که امکان ورود مستقیم را ندارند ، چندین پیش آیند وجود دارند کـه امکـان ورود کـاراکتر دلخـواه را فـراهم مینمایند، نظیر '\' برای ورود خود کاراکتر و '\n' برای کاراکتر خط جدیـد . همچنـین مکانیسـمی بـرای ورودی مسـتقیم مقـدار یـک کاراکتر در مبنای هشت یا شانزده وجود دارد . بـرای نشـانه گـذاری مبنـای هشـت از علامـت \ کـه توسـط یـک عـدد سـه رقمـی دنبـال

میشود،استفاده کنید. بعنوان مثال '141\' همـان حـرف 'a' اسـت . بـرای مبنـای شـانزده از علامـت (\u) اسـتفاده کنیـد و بعـد از آن دقیقا" چهار رقم مبنای شانزده . بعنوان مثال '\uoo61' که معادل حرف 'a' در استاندارد Iso-latin-1 است چون بایـت بـالایی آن صفر است '\ua432' . یک کاراکتر Katakana ژاپنی است . جدول زیر پیش آیندهای کاراکترها را نشان می دهد .

توصیف آن پیش آیند

- کاراکتر مبنای هشت (ddd) \ddd
- کاراکتر کد جهانی مبنای شانزده (xxxx) \uxxxx
- علامت تکی نقل قول '\0'
- علامت جفتی نقل قول | Backslash '\\'
- کاراکتر برگشت به سر خط \r
- خط جدید \n
- تغذیه فرم \t \f
- \b Backspace

متغیر String

متغیر رشته ای در جاوا نظیر سایر زبانهای برنامه نویسی مشخص می شوند قرار دادن یک دنباله از کاراکترها بین یـک جفـت از علامـات نقل قول ، در زیر نمونه هایی از متغیر رشته ای را مشاهده می کنید .

"Hello world"
"tow\nlines"
"\"This is in quotes\"

پیش آیندها و نشانه گذاریهای مبنای هشت / شانزده که برای متغیر کاراکترها توصیف شد ، بهمان روش در داخـل متغیـر رشـته ای کـار می کنند . یک نکته مهم درباره رشته های جاوا این است که آنها باید روی یک خط شروع شده و پایان یابد . بـرخلاف زبانهـای دیگـر در جاوا ادامه خط در خطهای دیگر مجاز نیست . نکته : حتما" می دانید که در اکثر زبانهای دیگر شامل C و C++ ، و رشته ها بعنـوان آرایه های کاراکتری پیاده سازی می شوند . اما در جاوا این حالت وجود نـدارد . رشـته هـا از نـوع اشـیائ هسـتند . بعـدا" مـی بینیـد از آنجاییکه جاوا پیاده سازی می کند .

انواع ساده The simple Types

جاوا هشت نوع ساده (یا ابتدایی) از داده را تعریف می کندint bbyte short : ، long،، char،، float،، double،، boolean، این انواع را می توان در چهار گروه بشرح زیر دسته بندی نمود :

integers اعداد صحیح:
این گروه دربرگیرندهbyte ، short،، int،وlong و میباشد که مختص ارقام علامتدار مقدار کل (whole-valued signed numbers)میباشد .

floating-point number اعداد اعشاری :
این گروه دربرگیرنده float وdouble است که معرف اعدادی است با دقت زیاد .

characters کاراکترها : (این گروه فقط شامل char بوده که نشانه هایی نظیر حروف و ارقام را در یک مجموعه خاص از کاراکترها معرفی می کند .

Boolean بولی : این گروه فقط شامل boolean است . که نوع خاصی ازمعرفی و بیان مقادیر صحیح / ناصحیح می باشد .

شما می توانید از این انواع همانطوریکه هستند استفاده کرده ، یا آرایه ها و انواع کلاسهای خود را بسازید . انواع اتمی معرف مقادیر تکی و نه اشیائ پیچیده هستند . اگر چه جاوا همواره شی ئ گرا است ، اما انواع ساده اینطور نیستند . این انواع ، مشابه انواع ساده ای هستند که در اکثر زبانهای غیر شی ئ گرا مشاهده می شود . دلیل این امر کارایی است . ساختن انواع ساده در اشیائ سبب افت بیش از حد کارایی و عملکرد می شود . انواع ساده بگونه ای تعریف شده اند تا یک دامنه روشن و رفتاری ریاضی داشته باشند . و زبانهایی نظیرC و++C و امکان می دهند تا اندازه یک عدد صحیح براساس ملاحظات مربوط به محیط اجرایی تغییر یابد . اما جاوا متفاوت عمل می کند . بدلیل نیازهای موجود برای قابلیت حمل جاوا ، کلیه انواع داده در این زبان دارای یک دامنه کاملا" تعریف شده هستند . بعنوان مثال یک int همیشه 32 بیتی است ، صرفنظر از اینکه زیر بنای خاص محیطی آن چگونه باشد . این حالت به برنامه های نوشته شده

اجازه می دهد تا با اطمینان و بدون در نظر گرفتن معماری خاص یک ماشین اجرا شوند. در حالیکه مشخص کردن دقیق اندازه یک عدد صحیح ممکن است در برخی محیط ها سبب افت عملکرد شود ، اما برای رسیدن به خاصیت قابلیت حمل پرداخت .

انواع اعداد اعشاری

اعداد اعشاری یا همان اعداد حقیقی برای ارزش گذاری عبارتهایی که نیازمند دقت بیشتری هستند ، استفاده می شوند . بعنوان نمونه ، محاسباتی نظیر ریشه دوم و محاسبات مثلثاتی نظیر سینوس و کسینوس منجربه جوابهایی می شوند که برای تعیین دقت آن نیاز به نوع عدد اعشاری می باشد . جاوا یک مجموعه استاندارد (IEEE-754) از انواع عدد اعشاری و عملگرها را پیاده سازی می کند. دو نوع عدد اعشاری تعریف شده یعنیfloat وdouble و هستند که بترتیب معرف دقت معمولی و مضاعف می باشند .

پهنا و دامنه آنها را در زیر نشان داده ایم :

دامنه پهنا بر حسب تعداد بیت نام

double 64 1.7e-308 to 1.7e+308

float 32 3.4e-038 to 3.4e+038

هر یک از انواع اعشاری را متعاقبا" مورد بررسی قرار می دهیم .

float

این نوع مشخص کننده یک مقدار با دقت معمولی بوده که از 32 بایت حافظه استفاده می کند . دقت معمول روی بعضی پردازنده ها سریعتر بوده و نسبت به دقت مضاعف نیمی از فضا را اشغال می کند ، اما هنگامیکه مقادیر خیلی بزرگ یا خیلی کوچک باشند ، دقت خود را از دست میدهد . متغیرهای نوع float برای زمانی مناسب هستند که از یک عضو کسری استفاده می کنید اما نیازی به دقت خیلی زیاد ندارید . بعنوان مثال ، نوع float برای معرفی دلار و سنت بسیار مناسب است float hightemp/ lowtemp; .

double

دقت مضاعف که با واژه کلیدی double معین می شود برای ذخیره کردن یک مقدار 64بیت فضا را اشغال می کند . دقت مضاعف روی برخی پردازنده های جدید که برای محاسبات ریاضی با سرعت زیاد بهینه شده اند ، واقعا" سریعتر از دقت معمولی عمل می کند . کلیه توابع مثلثاتی نظیر sin() ، cos()و sqrt() مقادیر مضاعف را برمی گردانند . هنگام اجرای محاسبات مکرر که نیاز به حفظ دقت دارید و یا هنگام کار با ارقام خیلی بزرگ double بهترین انتخاب است . در زیر برنامه ای را مشاهده می کنید که از doubleاستفاده نمود تا محیط یک دایره را محاسبه کند :

```
// Compute the area of a circle.
class Area {
public static void main(String args[] ){
double pi/ r/ a;
r = 10.8; // radius of circle
pi = 3.1416; // pi/ approximately
a = pi * r * r; // compute area
System.out.println("Area of circle is " + a);
}
}
```

کاراکترها

در جاوا از نوع داده char برای ذخیره کردن کاراکترها استفاده می شود . اما برنامه نویسانC وC++ و آگاه باشند که char در جاوا مشابه char در زبانهای C وC++ نیست . در زبانهایC وC++ و ، نوع char یک نوع عدد صحیح با پهنای 8 بیت است . اما جاوا متفاوت عمل می کند . جاوا از کدهای جهانی (unicode) برای معرفی کاراکترها استفاده می کند . کدهای جهانی یک مجموعه کاملا" جهانی از کاراکترها هستند که می توانند همه کاراکترها را معرفی نمایند . این مجموعه شامل دهها مجموعه کوچک تر کاراکتری نظیر Latin ، Greek، Arabic، Cyrillic، ، Hebrew، Katakana ، Hangul، و امثال آن است .

برای این منظور ، 16 بیت مورد نیاز است . بنابراین char در جاوا یک نوع 16 بیتی است . دامنه char از 0 تا 65/536 می باشد . در نوع char مقدار منفی وجود ندارد . مجموعه استاندارد کاراکترها موسوم به ASCII همچون گذشته دارای دامنه از 0 تا 127 و مجموعه کاراکترهای 8 بیتی توسعه یافته موسوم به Iso-Latin-1 دارای دامنه از 0 تا 255 می باشند . چون در جاوا امکان نوشتن ریز برنامه ها برای کاربری جهانی وجود دارد ، بنظر می رسد که بهتر است جاوا از کدهای جهانی برای معرفی کاراکترها استفاده نماید .

البته بکار بردن کدهای جهانی درمورد زبانهایی نظیر انگلیسی ،آلمانی ،اسپانیایی یا فرانسوی که کاراکترهای آنها را می توان براحتی داخل 8 بیت جای داد ، تا حدی سبب نزول کارآیی خواهد شد . اما این بهایی است که برای رسیدن به قابلیت حمل جهانی در برنامه ها باید پرداخت . نکته : اطلاعات بیشتر درباره کدهای جهانی را در آدرسهای وب زیر پیدا خواهید نمود :

http://www .unicode .org

http://www .stonehand .com/unicode .html

در زیر برنامه ای را مشاهده می کنید که متغیرهای char را نشان می دهد :

```
// Demonstrate char data type.
class CharDemo {
public static void main(String args[] ){
char ch1/ ch2;

ch1 = 88; // code for X
ch2 = 'Y';

System.out.print("ch1 and ch2 :");
System.out.println(ch1 + " " + ch2);
}
}
```

این برنامه خروجی زیر را نشان خواهد داد

```
: ch1 and ch2 :xy
```

دقت کنید که مقدار 88به ch1 نسبت داده شده ، که مقدار متناظر با حرف x در کد (ASCII و کد جهانی) است . قبلا" هم گفتیم که مجموعه کاراکتری ASCII ´127 مقدار اولیه در مجموعه کاراکتری کدهای جهانی را اشغال کرده است . بهمین دلیل کلیه فوت و فنهای قدیمی که قبلا" با کاراکترها پیاده کرده اید ، در جاوا نیز به خوبی جواب می دهند .

اگر چه انواع char عدد صحیح محسوب نمی شوند ، اما در بسیاری از شرایط می توانید مشابه عدد صحیح با آنها رفتار کنید . بدین ترتیب قادرید دو کاراکتر را با هم جمع نموده و یا اینکه مقدار یک متغیر کاركتری را کاهش دهید . بعنوان مثال ، برنامه زیر را در نظر بگیرید :

```
// char variables behave like integers.
class CharDemo2 {
public static void main(String args[] ){
char ch1;
ch1 = 'X';
System.out.println("ch1 contains " + ch1);
ch1++; // increment ch1
System.out.println("ch1 is now " + ch1);
}
}
```

خروجی این برنامه بشرح زیر خواهد بود

```
: ch1 contains x
ch1 is now y
```

در برنامه ابتدا مقدارx به ch1 داده میشود . سپس ch1 افزایش می یابد . این روال باعث می شود تا ch1 حرف y را اختیار کند، که کاراکتر بعدی در ترتیب ASCII و کدهای جهانی)می باشد .

boolean

جاوا یک نوع ساده موسوم به boolean برای مقادیر منطقی دارد . این نوع فقط یکی از مقادیر ممکنtrue یاfalse ا را اختیار می

کند . این نوعی است که توسط کلیه عملگرهای رابطه ای نظیر b شرطی که دستورهای کنترلی نظیر if و for و را مدیریت می کنند ، استفاده می شود .

در زیر برنامه ای مشاهده می کنید که نوع boolean را نشان می دهد :

```
// Demonstrate boolean values.
class BoolTest {
public static void main(String args[] ){
boolean b;
b = false;
System.out.println("b is " + b);
b = true;
System.out.println("b is " + b);
// a boolean value can control the if statement
if(b )System.out.println("This is executed.");
b = false;
if(b )System.out.println("This is not executed.");
// outcome of a relational operator is a boolean value
System.out.println("10 > 9 is " +( 10 > 9));
}
}
```

خروجی برنامه فوق بقرار زیر خواهد بود :

```
b is false
b is true
This is executed.
10>9 is true
```

درباره این برنامه سه نکته جالب توجه وجود دارد . اول اینکه وقتی که مقدار boolean توسط println() خارج می شود ، مـی بینیـد کـه "true" یـا "false" ا بنمـایش درمـی آیـد . دوم اینکـه یـک متغیـر boolean بتنهـایی بـرای کنتـرل دسـتور if کفایـت می کند . دیگر نیازی به نوشتن یک دستور if بقرار زیر نخواهد بود(if(b == true... :

یک مقدار> سوم اینکه ، پی آمد یک عملگر رابطه ای نظیر boolean است . بهمین دلیل است که عبارت 10>9 مقـدار true را نمایش می دهد . علاوه بر این ، مجموعه ی از پرانتزهایی که عبارت 10>9 را محصور کرده اند ، ضروری است زیرا عملگر

عملگرهای منطقی بولی boolean

عملگرهای منطقی بولی که در زیر نشان داده ایم فقط روی عملوندهای بولی عمل می کنند . کلیه عملگرهای منطقی باینری دو مقدار booleanرا ترکیب می کنند تا یک مقدار منتج boolean ایجاد نمایند .

نتیجه آن عملگر

AND

منطقی& OR

منطقی| XOR

منطقی (خارجOR ^)

مدار کوتاهAND ||

مدار کوتاهNOT &&

یکانی منطقی !

انتساب AND &=

انتساب OR |=

انتساب XOR ^=

مساوی با ==

نامساوی با !=

سه تایی ?: if-then-else

عملگرهای بولی منطقی& ، |،، ^، روی مقادیر Boolean همانطوری که روی بیت های یک عدد صحیح رفتار می کنند ، عمل خواهند کرد . عملگر منطقی ! حالت بولی را معکوس می کند :

!false=true t!true=false

جدول بعدی تاثیرات هر یک از عملیات منطقی را نشان می دهد :

A B A|B A&B A^B !A

False False False False False True

True False True False True False

False True True False True True

True True True True False False

در زیر برنامه ای را مشاهده می کنید که تقریبا" با مثال Bitlogic قبلی برابر است ، اما در اینجا بجای بیت های باینری روی مقادیر منطقی بولی عمل می کند :

```
// Demonstrate the boolean logical operators.
class BoolLogic {
public static void main(String args[] ){
boolean a = true;
boolean b = false;
boolean c = a | b;
boolean d = a & b;
boolean e = a ^ b;
boolean f =( !a & b )|( a & !b);
boolean g = !a;
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" a&b = " + d);
System.out.println(" a^b = " + e);
System.out.println("!a&b|a&!b = " + f);
System.out.println(" !a = " + g);
}
}
```

پس از اجرای این برنامه ، شما همان قوانین منطقی که برای بیت ها صادق بود در مورد مقادیر boolean مشاهده می کنید . در خروجی این برنامه مشاهده می کنید که معرفی رشته ای یک مقدار بولی درجاوا یکی از مقادیر لفظی true یا false است .

```
a = true
b = false
a|b = true
a&b = false
```

```
a^b = true
!a&b|a&!b = true
!a = false
```

عملگرهای منطقی مدار کوتاه

جاوا دو عملگر بولی بسیار جالب دارد که در اکثر زبانهای دیگر برنامه نویسی وجود ندارند . این ها روایت ثانویه عملگرهـایAND و ORو بولی هستند و بعنوان عملگرهای منطقی مدار کوتاه معرفی شده اند. در جدول قبلی می بینید که عملگر OR هرگاه که A معادل trueباشد، منجر به true می شود، صرفنظر از اینکه B چه باشد .بطور مشابه ، عملگر AND هرگاه A معادل false باشد منجر به falseمی شود. صرفنظر از اینکه B چه باشد . اگر از اشکال ||و&& و بجای |و& و استفاده کنید، هنگامیکه حاصل یک عبارت مـی تواند توسط عملوند چپ بتنهایی تعیین شود ، جاوا دیگر به ارزیابی عملوند راست نخواهـد پرداخـت . ایـن حالـت بسیار سـودمند است بخصوص وقتی که عملوند سمت راست بستگی به عملوند سمت چپ وtrue یـاfalse ا بـودن آن بـرای درسـت عمـل کـردن داشـته باشد . بعنوان مثال ، کد قطعه ای زیر به شما نشان می دهد چگونه می توانید مزایای ارزیابی منطقی مدار کوتاه را استفاده نموده تا مطمئن شوید که عملیات تقسیم قبل از ارزیابی آن معتبر است .

```
if(denom != 0 && num / denom > 10)
```

از آنجاییکه شکل مدار کوتاه AND یعنی && استفاده شده است ، هنگامیکه denom صفر باشد ، خطر ایجـاد یـک اسـتثنائ حـین اجرا منتفی است . اگر همین خط از کد را با اسـتفاده از روایـت تکـی AND یعنـی & بنویسـیم ، هـر دو عملونـد بایـد مـورد ارزیـابی قرار گیرند و هنگامیکه denom صفر باشد یک استثنائ حین اجرا بوجود می آید . در حالتهایی که شامل منطق بولی باشند : اسـتفاده از ارزیابیهای مدار کوتاهAND وOR و یک روش استاندارد است که روایتهای تک کـاراکتری عملگرهـا را منحصـرا "بـرای عمليـات رفتار بیتی قرار می دهد . اما استثنائاتی بر این قوانین وجود دارند . بعنوان مثال ، دستور زیر را در نظر بگیرید :

```
if(c==1 & e++ < 100 )   d = 100;
```

در اینجا استفاده از یک علامت & تکی اطمینان می دهد که عملیات افزایشی بهe

اعلان نمودن اشیائ

بدست آوردن اشیائ از یک کلاس ، نوعی پردازش دو مرحله ای است . اول ، باید یک متغیر از نوع همـان کـلاس اعـلان نماییـد . ایـن متغیر یک شی ئ را تعریف نمی کند . در عوض ، متغیری است که می تواند به یک شی ئ ارجاع نماید .دوم ، باید یک کپی فیزیکـی

و واقعی از شی ئ بدست آورده و به آن متغیر منتسب کنید . می توانید اینکار را با استفاده از عملگر new انجام دهید . عملگر new بطور پویا (یعنی در حین اجرا) حافظه را برای یک شی ئ تخصیص داده و یک ارجاع به آن را برمی گرداند . این ارجاع (reference)کمابیش آدرس آن شی ئ در حافظه است که توسط new تخصیص یافته است . سپس این ارجاع در متغیر ذخیره می شود. بدین ترتیب ، در جاوا، کلیه اشیائ کلاس دار باید بصورت پویا تخصیص یابند. اجازه دهید که به جزئیات این روال دقت نماییم . در مثال قبلی ، یک خط مشابه خط زیر برای اعلان یک شی ئ ازنوع Box استفاده شده

```
Box mybox = new Box();
```

این دستور دو مرحله گفته شده را با یکدیگر ترکیب نموده است . برای اینکه هر یک از مراحل را روشن تر درک کنید، میتوان آن دستور را بصورت زیر بازنویسی نمود :

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

خط اول ، myBoxرا بعنوان یک ارجاع به شیئی از نوع Box اعلان می کند . پس از اجرای این خط ، myboxمحتوی تهی (null)خواهد داشت که نشانگر آن است که هنوز شی ئ بطور واقعی بوجود نیامده است . هر تلاشی برای استفاده از myboxدر این مرحله سبب بروز خطای زمان کامپایل (compile-time error) خواهد شد . خط بعدی یک شی ئ واقعی را تخصیص داده و یک ارجاع از آن به mybox انجام می دهد . پس از اجرای خط دوم ، می توانید از mybox بعنوان یک شی ئ Box استفاده نمایید . اما در واقعیتmybox خیلی ساده آدرس حافظه شی ئ واقعی Box را نگهداری می کند . تاثیر این دو خط کد را در شکل زیر نشان داده ایم .

نکته : کسانی که با C++/C آشنایی دارند احتمالا" توجه نموده اند که ارجاعات شی ئ مشابه اشاره گرها هستند . این تشابه تا حدود زیادی صحیح است . یک ارجاع شی ئ (object reference) مشابه یک اشاره گر حافظه است . مهمترین تفاوت و کلید ایمنی در جاوا این است که نمی توانید از ارجاعات همچون اشاره گرهای واقعی استفاده نمایید . بدین ترتیب ، نمی توانید ارجاع شی ئ را بعنوان اشاره ای به موقعیت دلخواه حافظه یا بعنوان یک عدد صحیح بکار برید .

Statement Effect
Box mybox; //(null)
mybox

mybox Width = new Box ()

mybox Height

Depth

Box object

نگاهی دقیقتر به new

شکل عمومی عملگر new بقرار زیر می باشد :

```
class-var = new classname();
```

در اینجا class-var یک متغیر از نوع کلاسی است که ایجاد کرده ایم . class name نام کلاسی است که می خواهیم معرفی کنیم . نام کلاس که بعد از آن پرانتزها قرار گرفته اند مشخص کننده سازنده (constructor) کلاس است . سازنده تعریف می کند که وقتی یک شی ئ از یک کلاس ایجاد شود ، چه اتفاقی خواهد افتاد . سازنده ها بخش مهمی از همه کلاسها بوده و خصلتهای بسیار قابل توجهی دارند . بسیاری از کلاسهای دنیای واقعی (real-world) بطور صریحی سازندگان خود را داخل تعریف کلاس ، معرفی می کنند . اما اگر سازنده صریحی مشخص نشده باشد ، جاوا بطور خودکار یک سازنده پیش فرض را عرضه می کند . درست مثل حالت Box . در این مرحله ، ممکن است تعجب کنید که چرا از new برای مواردی نظیر اعداد صحیح وکاراکترها استفاده نمی شود. جواب این است که انواع ساده در جاوا بعنوان اشیائ پیاده سازی نمی شوند . در عوض ، آنها بعنوان متغیرهای عادی پیاده سازی می شوند. اینکار برای افزایش کارایی انجام می گیرد. جاوا قادراست بدون استفاده از رفتارهای خاص نسبت به اشیائ، این انواع ساده را بطور موثری پیاده سازی کند .نکته مهم این است که new حافظه را برای یک شی ئ طی زمان اجرا تخصیص می دهد .

مزیت این روش آن است که برنامه شما میتواند اشیائ مورد نیازش را طی زمان اجرای برنامه ایجاد کند . اما از آنجاییکه محدودیت حافظه وجود دارد ، ممکن است new بعلت عدم کفایت حافظه نتواند حافظه را به یک شی ئ تخصیص دهد . اگر چنین حالتی پیش بیاید، یک استثنائ حین اجرا واقع خواهد شد .ولی در زبانهای C/C++ در صورت عدم موفقیت ، مقدار تهی (null) برگردان می شود .

اجازه دهید یکبار دیگر تفاوت بین یک کلاس و یک شی ئ را مرور کنیم . یک کلاس یک نوع جدید داده را ایجاد می کند که می توان برای تولید اشیائ از آن نوع استفاده نمود. یعنی یک کلاس یک چهارچوب منطقی ایجاد می کند که ارتباط بین اعضائ را توصیف می نماید . هنگامیکه یک شی ئ از یک کلاس را اعلان می کنید ، در حقیقت نمونه ای از آن کلاس را بوجود آورده اید. بدین ترتیب ، کلاس یک ساختار منطقی است . یک شی دارای واقعیت فیزیکی است . (یعنی یک شی ئ فضایی از حافظه را اشغال در ذهن داشته باشید .

عملگرها

جاوا یک محیط عملگر غنی را فراهم کرده است . اکثر عملگرهای آن را می توان در چهار گروه طبقه بندی نمود :حسابی arithmetic رفتار بیتی bitwise رابطه ای ralational و منطقی logical جاوا همچنین برخی عملگرهای اضافی برای اداره حالتهای خاص و مشخص تعریف کرده است . نکته : اگر با C/C++ آشنایی دارید ، حتما" خوشحال می شوید که بدانید کارکرد عملگرها در جاوا دقیقا"مشابه با C/C++ است . اما همچنان تفاوتهای ظریفی وجود دارد .

عملگرهای حسابی Arithmetic operators

عملگرهای حسابی در عبارات ریاضی استفاده مـی شـوند و طریقـه استـفاده از آنهـا بهمـان روش جبـری اسـت . جـدول بعـدی فهرسـت عملگرهای حسابی را نشان می دهد :

نتیجه آن عملگر

اضافه نمودن +

تفریق نمودن : همچنین منهای یکانی

ضرب *

تقسیم /

تعیین باقیمانده %

افزایش ++

انتساب اضافه نمودن +=

انتساب تفرق نمودن -=

انتساب ضرب نمودن *=

انتساب تقسیم نمودن /=

انتساب تعیین باقیمانده %=

کاهش - -

عملوندهای مربوط به عملگرهای حسابی باید از نوع عددی باشند . نمی توانید از این عملگرها روی نوع boolean استفاده کنید ، اما روی انـواع char قابـل اسـتفاده هسـتند ، زیـرا نـوع char در جـاوا بطـور ضـروری زیـر مجموعـه ای از int اسـت .

عملگرهای اصلی حسابی

عملیات اصلی حسابی جمع ، تفریق ، ضرب و تقسیم همانطوریکه انتظار دارید برای انواع عددی رفتار مـی کننـد . عملگـر تفـرق نمـودن همچنین یک شکل یکانی دارد که عملوند تکی خود را منفی (یا خنثی) می کند . بیاد آورید هنگامیکه عملگر تقسیم به یک نوع عـدد صحیح اعمال می شود ، هیچ عنصری کسری یا خرده به جواب ملحق نمی شود . برنامه ساده بعدی نشاندهنده عملگرهای حسابی است . این برنامه همچنین تفاوت بین تقسیم اعشاری و تقسیم عدد صحیح را توضیح می دهد .

```
// Demonstrate the basic arithmetic operators.
```

```
class BasicMath {
public static void main(String args[] ){
// arithmetic using integers
System.out.println("Integer Arithmetic");
int a = 1 + 1;
int a = a * 3;
int a = b / 4;
int a = c - a;
int a =- d;
System.out.println("a = " + a);
System.out.println("a = " + b);
System.out.println("a = " + c);
System.out.println("a = " + d);
System.out.println("a = " + e);
// arithmetic using doubles
System.out.println("\nFloating Point Arithmetic");
double da = 1 + 1;
double db = da * 3;
double dc = db / 4;
double dd = dc - a;
double de =- dd;
System.out.println("da = " + da);
System.out.println("db = " + db);
System.out.println("dc = " + dc);
System.out.println("dd = " + dd);
System.out.println("de = " + de);
}
}
```

خروجی این برنامه بقرار زیر می باشد :

```
integer Arithmetic
a=2
b=6
c=1
d=-1
e=1
```

```
floating point arithmetic
da=2
db=6
dc=1.5
dd=-0.5
de=0.5
```

عملگر تعیین باقیمانده The Modulus operator

عملگر تعیین باقیمانده یعنی % ، باقیمانده یک عملیات تقسیم را برمی گرداند .این عملگر برای انواع عدد اعشاری و انواع عدد صحیح قابل استفاده است . (اما در C++/C این عملگر فقط در مورد انواع عدد صحیح کاربرد دارد .) برنامه بعدی نشان دهنده عملگر ٪ می باشد :

```
// Demonstrate the % operator.
class Modulus {
public static void main(String args[] ){
int x = 42;
double y = 42.3;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}
}
```

هنگامیکه این برنامه را اجرا می کنید ، خروجی زیر حاصل می شود :

```
x mod 10=2
y mod 10=2.3
```

عملگرهای انتساب حسابی Arithmetic Assignment operators

جاوا عملگرهای ویژه ای را تدارک دیده که با استفاده از آنها می توان یک عملیات حسابی را با یک انتساب ترکیب نمود . احتمالا" می دانید که دستوراتی نظیر مورد زیر در برنامه نویسی کاملا" رایج هستند :

```
a = a + 4;
```

در جاوا ، می توانید این دستور را بصورت دیگری دوباره نویسی نمایید :

```
a += 4;
```

این روایت جدید از عملگر انتساب += استفاده می کند هر دو دستورات یک عمل واحد را انجام می دهند : آنها مقدار a را 4 واحد افزایش می دهند . اکنون مثال دیگری را مشاهده نمایید :

```
a = a % 2;
```

که می توان آن را بصورت زیر نوشت :

```
a %= 2;
```

در این حالت ٪= باقیمانده a/2 را گرفته و حاصل را مجددا" در a قرار می دهد. عملگرهای انتسابی برای کلیه عملگرهای حسابی و دودوئی (باینری) وجود دارند . بنابراین هر دستور با شکل :

```
Var = var op expression;
```

عبارت عملگر متغیر متغیر را می توان بصورت زیر دوباره نویسی نمود :

```
var op = expression;
```

عبارت عملگر متغیر عملگرهای انتساب دو مزیت را بوجود می آورند . اول اینکه آنها یک بیت از نوع بندی را برای شما صرفه جویی می کنند ، زیر آنها کوتاه شده شکل قبلی هستند. دوم اینکه آنها توسط سیستم حین اجرای جاوا بسیار کاراتر از اشکال طولانی خود پیاده سازی می شوند. بهمین دلایل ، در اکثر برنامه های حرفه ای نوشته شده با جاوا این عملگرهای انتساب را مشاهده می کنید .

در زیر برنامه ای وجود دارد که چندین عملگر انتساب op را نشان می دهد :

```
// Demonstrate several assignment operators.
class OpEquals {
public static void main(String args[] ){
int a = 1;
int b = 2;
int c = 3;
a += 5;
b *= 4;
```

```
c += a * b;
c %= 6;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
}
}
```

خروجی این برنامه بقرار زیر می باشد :

```
a=6
b=8
c=3
```

افزایش و کاهش Increment and Decrement

علامات ++ و –– عملگرهای افزایشی و کاهشی جاوا هستند . این عملگرها را قبلا "معرفی کرده ایم . در اینجا آنها را با دقت بیشتری بررسی می کنیم . همانگونه که خواهید دید ، این عملگرها خصلتهای ویژه ای دارند که بسیار جالب توجه است . بحث درباره این عملگرها را از نحوه کار آنها شروع می کنیم . عملگر افزایشی ، عملوند خود را یک واحد افزایش می دهد . عملگر کاهشی نیز عملوند خود را یک واحد کاهش می دهد . بعنوان مثال ، دستور زیر را

```
x = x + 1;
```

می توان با استفاده از عملگر افزایشی بصورت زیر دوباره نویسی نمود :

```
x++;
```

بطور مشابهی ، دستور زیر را

```
x = x - 1;
```

می توان بصورت زیر باز نویسی نمود :

```
x--;
```

این عملگرها از آن جهت که هم بشکل پسوند جایی که بعد از عملوند قرار می گیرند و هم بشکل پیشوند جایی که قبل از عملوند قرار می گیرند ظاهر می شوند کاملا" منحصر بفرد هستند. در مثالهای بعدی هیچ تفاوتی بین اشکال پسوندی و پیشوندی وجود ندارد . اما

هنگامیکه عملگرهای افزایشی و کاهشی بخشی از یک عبارت بزرگتر هستند ، آنگاه یک تفاوت ظریف و در عین حال پرقدرت بین دو شکل وجود خواهد داشت . در شکل پیشوندی ، عملوند قبل از اینکه مقدار مورد استفاده در عبارت بدست آید ، افزایش یا کاهش می یابد . در شکل پسوندی ، ابتدا مقدار استفاده در عبارت بدست می آید ، و سپس عملوند تغییر می یابد . بعنوان مثال:

```
x = 42;
y = ++x;
```

در این حالت ، همانطوریکه انتظار دارید y معادل 43 می شود ، چون افزایش قبل از اینکه x به y منتسب شود ، اتفاق می افتد . بدین ترتیب خط y=++ معادل دو دستور زیر است :

```
x = x + 1;
y = x;
```

اما وقتی که بصورت زیر نوشته می شوند :

```
x = 42;
y = x++;
```

مقدار x قبل از اینکه عملگر افزایشی اجرا شود ، بدست می آید ، بنابراین مقدار y معادل 42 می شود . البته در هر دو حالت x معادل 43 قرار می گیرد . در اینجا ، خط y=x++ معادل دو دستور زیر است :

```
y = x;
x = x + 1;
```

برنامه بعدی نشان دهنده عملگر افزایشی است .

```
// Demonstrate ++.
class IncDec {
public static void main(String args[] ){
int a = 1;
int b = 2;
int c;
int d;
c = ++b;
d = a++;
```

```
c++;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
}
}
```

خروجی این برنامه بقرار زیر می باشد :

```
a=2
b=3
c=4
d=1
```

عملگرهای رفتار بیتی The Bitwise operators جاوا چندین عملگر رفتار بیتی تعریف نموده که قابل اعمال روی انواع عدد صحیح شامل long ، int، short، char وbyte و می باشند . این عملگرها روی بیت های تکی عملوندهای خود عمل می کنند .

این عملگرها را در جدول زیر خلاصه نموده ایم :

نتیجه آن عملگر

Bitwise unary Not

Notیکانی رفتار بیتی Bitwise AND ~

ANDرفتار بیتی Bitwise OR &

ORرفتار بیتی Bitwise exclusive OR |

ORخارج رفتار بیتی shift right ^

>>حرکت براست shift right zero fill

>>>حرکت براست پر شده با صفر shift left

<<حرکت به چپ Bitwise AND assignment

انتساب AND رفتار بیتی Bitwise OR assignment =&

انتساب OR رفتار بیتی Bitwise exclusive OR assignment =|

انتساب OR خارج رفتار بیتی shift right assignment =^

انتساب حرکت راستshift right zero fill assignment<< =

انتساب حرکت براست پر شده با صفرshift left assignment<<< =

انتساب حرکت به چپ=<<

از آنجاییکه عملگرهای رفتار بیتی با بیت های داخل یک عدد صحیح سر و کار دارند ، بسیار مهم است بدانیم که این سر و کار داشتن چه تاثیری ممکن است روی یک مقدار داشته باشد . بخصوص بسیار سودمند است بدانیم که جاوا چگونه مقادیر عدد صحیح را ذخیره نموده و چگونه اعداد منفی را معرفی می کند . بنابراین ، قبل از ادامه بحث ، بهتر است این دو موضوع را باختصار بررسی نماییم . کلیه انواع صحیح بوسیله ارقام دودوئی (باینری) دارای پهنای بیتی گوناگون معرفی میشوند. بعنوان مثال ، مقدار byte عدد 42 در سیستم باینری معادل 00101010 است ، که هر یک از این نشانه ها یک توان دو را نشان می دهند که با 2 به توان 0 در بیت سمت راست شروع شده است . یا موقعیت بعدی بیت بطرف چپ 2'یا 2ا است و به طرف چپ بیت بعدی 2 به توان 2یا4 ا است ، بعدی 8 ، 16 ، 32 و همینطور الی آخر هستند . بنابراین عدد 42 بیت 1 را در موقعیتهای اول ، سوم و پنجم (از سمت راست درنظربگیرید) دارد. بدین ترتیب 42 معادل جمع 5 بتوان 3+2 بتوان 1+2 بتوان 2 یعنی 2+8+32 می باشد . کلیه انواع عدد صحیح (باستثنائ char) اعداد صحیح علامت دار هستند . یعنی که این انواع مقادیر منفی را همچون مقادیر مثبت می توانند معرفی کنند . جاوا از یک روش رمزبندی موسوم به مکمل دو (two's complement) استفاده می کند که در آن ارقام منفی با تبدیل (تغییر 1به 0 و بالعکس) کلیه بیت های یک مقدار و سپس اضافه نمودن 1 به آن معرفی می شوند . بعنوان مثال برای معرفی 42 ، ابتدا کلیه بیت های عدد 42 (00101010) را تبدیل می نماییم که 11010101 حاصل می شود آنگاه 1 را به آن اضافه می کنیم . که حاصل نهایی یعنی 11010110 معرف عدد 42 خواهد بود . برای رمز گشایی یک عدد منفی ، کافی است ابتدا کلیه بیت های آن را تبدیل نموده ، آنگاه 1 را به آن اضافه نماییم . 42- یعنی 11010110 پس از تبدیل برابر 00101001 یا 41 شده و پس از اضافه نمودن 1 به آن برابر 42 خواهد شد . دلیل اینکه جاوا)واکثر زبانهای برنامه نویسی)از روش مکمل دو (two's complement)استفاده می کنند ، مسئله تقاطع صفرها (Zero crossing) است . فرض کنید یک مقدارbyte برای صفر با 00000000 معرفی شده باشد. در روش مکمل یک (one's complement) تبدیل ساده کلیه بیت ها منجر به 11111111 شده که صفر منفی را تولید می کند

.

اما مشکل این است که صفر منفی در ریاضیات عدد صحیح غیر معتبر است . این مشکل بااستفاده از روش مکمل دو (two's complement)برای معرفی مقادیر منفی حل خواهد شد . هنگام استفاده از روش مکمل دو ، 1 به مکمل اضافه شده و عدد 100000000 تولید می شود . این روش بیت 1 را در منتهی الیه سمت چپ مقدار byte قرار داده تا رفتار مورد نظر انجام گیرد، جایی که 0با 0ا یکسان بوده و 11111111 رمزبندی شده 1 است . اگر چه در این مثال از یک مقدار byte استفاده کردیم ، اما

همین اصول برای کلیه انواع عدد صحیح جاوا صدق می کنند . از آنجاییکه جاوا از روش مکمل دو برای ذخیره سازی ارقام منفی استفاده میکند و چون کلیه اعداد صحیح در جاوا مقادیر علامت دار هستند بکار بردن عملگرهای رفتار بیتی براحتی نتایج غیر منتظره ای تولید می کند . بعنوان مثال برگرداندن بیت بالاتر از حد مجاز (high-order) سبب می شود تا مقدار حاصله بعنوان یک رقم منفی تفسیر شود ، خواه چنین قصدی داشته باشید یا نداشته باشید . برای جلوگیری از موارد ناخواسته ، فقط بیاد آورید که بیت بالاتر از حد مجاز (high-order)

علامت یک عدد صحیح را تعیین می کند، صرفنظر از اینکه بیت فوق چگونه مقدار گرفته باشد .

عملگرهای منطقی رفتار بیتی

عملگرهای منطقی رفتار بیتی شامل& ، |،، ^،، ~، هستند. جدول زیر حاصل هر یک از این عملیات را نشان می دهد . در بحث بعدی بیاد داشته باشید که عملگرهای رفتار بیتی به بیت های منفرد داخل هر عملوند اعمال می شوند .

A	B	A|B	A&B	A^B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

NOT

رفتار بیتی

عملگر NOT یکانی یعنی ~ که آن را مکمل رفتار بیتی (bitwise complement) هم می نامند ، کلیه بیت های عملوند خود را تبدیل می کند . بعنوان مثال ، عدد 42 که مطابق الگوی بیتی زیر است : 00101010 پس از اعمال عملگر NOT بصورت زیر تبدیل می شود11010101 :

رفتار بیتیAND

عملگر AND یعنی & اگر هر دو عملوند 1 باشند ، یک بیت 1 تولید می کند . در کلیه شرایط دیگر یک صفر تولید می شود . مثال زیر را نگاه کنید : 42 00101010

& 00001111 15

00001010 10

رفتار بیتیOR

عملگر OR یعنی | بیت ها را بگونه ای ترکیب می کند که اگر هر یک از بیت های عملوندها 1 باشد ، آنگاه بیت حاصله نیز 1 خواهد بود . به مثال زیر نگاه کنید : 42 00102010

00001111 15

00101111 47

رفتار بیتیXOR

عملگر XOR یعنی ^ بیت ها را بگونه ای ترکیب می کند که اگر دقیقا" یک عملوند 1باشد ، حاصل برابر 1 خواهد شد . در غیر اینصورت ، حاصل 0 می شود . مثال بعدی چگونگی کار این عملگر را نشان می دهد . این مثال همچنین یکی از خصلتهای سودمند عملگر XOR را نمایش می دهد . دقت کنید که هر جا عملوند دوم یک بیت 1 داشته باشد ، چگونه الگوی بیتی عدد 42 تبدیل می شود . هر جا که عملوند دوم بیت 0 داشته باشد ، عملوند اول بدون تغییر می ماند . هنگام انجام برخی از انواع عملکردهای بیتی ، این خصلت بسیار سودمند است . 4200101010

^ 00001111 15

00100101 37

استفاده از عملگرهای منطقی رفتار بیتی

برنامه بعدی نشان دهنده عملگرهای منطقی رفتار بیتی است :

```
// Demonstrate the bitwise logical operators.
class BitLogic {
public static void main(String args[] ){
String binary[] = {
"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
"1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
};
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a & b;
int e = a ^ b;
int f =( ~a & b )|( a & ~b);
```

```
int g = ~a & 0x0f;

System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);
System.out.println(" a|b = " + binary[c]);
System.out.println(" a&b = " + binary[d]);
System.out.println(" a^b = " + binary[e]);
System.out.println("~a&b|a&~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);
}
}
```

در این مثال ، aوb و الگوهای بیتی دارند که کلیه چهار احتمال برای ارقام دو تایی بـاینری را معرفی مـی کننـد . 0-0، 0-1، 1-0،و 1-1و . می توانید مشاهده کنید چگونه |و& و روی هر یک از بیت ها با توجه به نتایج درc و d و عمل مـی کننـد. مقـادیر نسبت داده شده بهe و f و مشابه بوده و نشان دهنده چگونگی کار عملگر ^ می باشند . آرایه رشـته ای بـا نـام binary معرفـی ارقـام 0 تـا 15 را بصورت باینری و قابل خواندن برای انسان نگهداری می کند . در این مثال ، آرایه فوق طوری نمایه سازی شده تا معرفی باینری هر یـک از نتایج را نشان دهد . آرایه طوری ساخته شده که معرفی رشته ای صحیح یک مقدار بـاینری n را در binary[n] ذخیـره مـی کنـد . مقدارa~ بوسیله عملگر AND با oxof (00001111 باینری) عمل شده تا مقدار آن را بـه کمتـر از 16 کـاهش دهـد تـا بتـوان بـا استفاده از آرایه binary از آن چاپ گرفت . اکنون خروجی این برنامه بصورت زیر می باشد :

```
a=1011
b=0110
a^Eb=0111
a&b=0010
a&b=0101
~a&b^Ea&~b=0101
~a=1100
```

حرکت به چپ

کلیه بیت های موجود در یک مقدار را به تعداد>> عملگر حرکت به چپ یعنی دفعات مشخص بطرف چپ منتقل می کند . شکل کلی آن بقرار زیر است :

```
Value << num
```

تعداد دفعات مقدار

در ایجا num مشخص کننده تعداد مکانهایی است که بیت های موجود در value باید کلیه بیت های موجود در یک مقدار مشخص را<< به چپ انتقال یابند . بدین ترتیب بتعداد مکانهایی که در num مشخص شده بطرف چپ حرکت می دهد . برای هر بار حرکت به چپ ، بیت (high-order بیش از حد مجاز) منتقل شده و از دست خواهد رفت و یک صفر در طرف راست مقدار ، جایگزین می شود . بدین ترتیب هنگامیکه یک حرکت به چپ روی یک عملوند int عمل می کند ، بیت های گذشته از مکان 31 از دست خواهند رفت . اگر عملوند یک long باشد، بیت ها پس از گذشتن از مکان 63 از دست میروند .هنگامیکه مقادیرbyte و وshort را انتقال می دهید ، ارتقائ خودکار انواع در جاوا نتایج غیر منتظره ای ایجاد می کند . حتما" می دانید که هنگام ارزشیابی عبارات ، مقادیرbyte وshort وبه int ارتقائ می یابند. بعلاوه جواب چنین عبارتی از نوع int خواهد بود .بنابراین حاصل یک حرکت به چپ روی مقادیرbyte وshort و یک int خواهد بود و بیت های انتقال یافته به چپ تا زمانیکه از مکان بیت 31 نگذرند ، از دست نمی روند . علاوه براین ، یک مقدار منفیbyte وshort و هنگامیکه به int ارتقائ می یابد ، بسط علامت پیدا می کند . بنابراین بیت های بیش از حد مجاز با بیت 1 پر می شوند . بخاطر این دلایل ، انجام یک حرکت به چپ روی byte وshort مستلزم آن است که از بایت های بیش از حد مجاز در جواب int دست بکشید . بعنوان مثال ، اگر یک مقدار byte را حرکت به چپ بدهید ، آن مقدار ابتدا به نوعint تبدیل شده و سپس انتقال خواهد یافت . باید سه بایت بالایی حاصل را از دست بدهید . اگر بخواهید حاصل یک مقدار byte انتقال یافته را بدست آورید . باید سه بایت بالایی حاصل را از دست بدهید . آسان ترین روش برای انجام اینکار استفاده از تبدیل cast و تبدیل جواب به نوع byte است .

مثال بعدی همین مفهوم را برای شما آشکار می سازد :

```
// Left shifting a byte value.
class ByteShift {
public static void main(String args[] ){
byte a = 64/ b;
int i;

i = a << 2;
b =( byte( )a << 2);

System.out.println("Original value of a :" + a);
System.out.println("i and b :" + i + " " + b);
}
}
```

خروجی تولید شده توسط این برنامه بقرار زیر می باشد :

```
original value of a:64
i and b :256 0
```

چون برای اهداف ارزشیابی ، aبه نوع int ارتقائ یافته ، دوبار حرکت به چپ مقدار 64 (0100 0000) منجر به i می گردد که شامل مقدار 256 (1 0000 0000)می باشد . اما مقدار b دربرگیرنده صفر است زیرا پس از انتقال ، بایت کمتر از حد مجاز (loworder) اکنون شامل صفر است . تنها بیت دربرگیرنده 1 به بیرون انتقال یافته است .

از آنجاییکه هر بار حرکت به چپ تاثیر دو برابر سازی مقدار اصلی را دارد برنامه نویسان اغلب از این خاصیت بجای دو برابر کردن استفاده می کنند . اما باید مراقب باشید . اگر یک بیت 1 را به مکان بیت بیش از حد مجاز (31 یا 63) منتقل کنید ، مقدار فوق منفی خواهد شد . برنامه بعدی همین نکته را نشان میدهد .

```
// Left shifting as a quick way to multiply by 2.
class MultByTwo {
public static void main(String args[] ){
int i;
int num = 0xFFFFFFE;

for(i=0; i<4; i++ ){
num = num << 1;
System.out.println(num);
}
}
}
```

خروجی این برنامه بقرار زیر خواهد بود :

```
536870908

1073741816
2147483632
- 32
```

مقدار آغازین را با دقت انتخاب کرده ایم بطوریکه بیت بعد از چهار مکان حرکت بطرف چپ ، مقدار 32 -را تولید نماید .

همانطوریکه می بینید ، هنگامیکه بیت 1 به بیت 31 منتقل می شود ، رقم بعنوان منفی تفسیر خواهد شد .

حرکت به راست

کلیه بیت های موجود در یک مقدار را به تعداد>> عملگر حرکت به راست یعنی دفعات مشخص بطرف راست انتقال می دهد . شکل کلی آن بقرار زیر می باشد :

```
value >> num
```

تعداد دفعات مقدار

در اینجا ، numمشخص کننده تعداد مکانهایی است که بیت های value باید بطرف کلیه بیت های یک مقدار مشخص شده را به تعداد>> راست انتقال یابند . یعنی مکانهای بیتی مشخص شده توسط num بطرف راست انتقال می دهد . کد قطعه ای زیر مقدار 32 را دو مکان بطرف راست منتقل می کند و آنگاه جواب آن در a معادل 8 قرار می گیرد :

```
int a = 32;
a = a >> 2; // a now contains 8
```

اگر بیت هایی از یک مقدار به بیرون منتقل شوند ، آن بیت ها از دست خواهند رفت . بعنوان مثال کد قطعه ای بعدی مقدار 35 را دو مکان بطرف راست منتقل نموده و باعث می شود تا دو بیت کمتر از حد مجاز از دست رفته و مجددا "جواب آن در a معادل 8 قرار گیرد .

```
int a = 35;
a = a >> 2; // a still contains 8
```

همین عملیات را در شکل باینری نگاه می کنیم تا اتفاقی که می افتد ، روشن تر شود : 35 00100011

>> 2

00001000 8

هر بار که یک مقدار را به طرف راست منتقل می کنید ، آن مقدار تقسیم بر دو می شود و باقیمانده آن از دست خواهد رفت . می توانید از مزایای این روش در تقسیم بر دو اعداد صحیح با عملکرد سطح بالا استفاده نمایید . البته ، باید مطمئن شوید که بیت های انتهایی سمت راست را به بیرون منتقل نکنید . هنگامیکه حرکت بطرف راست را انجام می دهید ، بیت های بالایی (از سمت چپ) در معرض حرکت بطرف راست قرار گرفته ، با محتوی قبلی بیت بالایی پر می شوند . این حالت را بسط علامت (sign extension) نامیده و برای محفوظ نگهداشتن علامت ارقام منفی هنگام حرکت بطرف راست استفاده می شوند. بعنوان مثال 1 -8>>معادل 4 -است که بـه شکل باینری زیر می باشد : 11111000 -8

>> 111111100- 4

جالب است بدانید که اگر 1- را بطرف راست حرکت دهید، حاصل آن همواره 1- باقی می ماند ، چون بسط علامـت ، مراقـب آوردن یک بیت دیگر در بیت های بیش از حد مجاز خواهد بود . گاهی هنگام حرکت بطرف راست مقادیر ، مایل نیستیم تا بسط علامت اجـرا شود . بعنوان مثال ، برنامه بعدی یک مقدار نوع byte را به معرفی رشته ای در مبنای 16 تبدیل می کند . دقت کنید که مقـدار منتقـل شده با استفاده از عملگر AND یا oxof ا پوشانده شده تا هر گونه بیت های بسط یافته علامت را بـدور انـدازد بطوریکـه مقـدار فـوق را بتوان بعنوان یک نمایه به آرایه ای از کاراکترهای در مبنای 16 استفاده نمود .

```
// Masking sign extension.
class HexByte {
static public void main(String args[] ){
char hex[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f' };
byte b =( byte )oxf1
System.out.println("b = ox" + hex[(b >> 4 )& oxof] + hex[b & oxof]);
}
}
```

خروجی این برنامه بقرار زیر می باشد :

```
b=oxf1
```

حرکت به راست فاقد علامت بطور خودکار>> اکنون می دانید که هر بار یک انتقال اتفاق می افتد ، عملگر جای خالی بیت بیش از حد مجاز را با محتوی قبلی اش پر می کند . این عمل سبب حفظ علامت آن مقدار می گردد . اما گاهی تمایلی برای اینکار نداریم .

بعنوان مثال

می خواهید چیزی را منتقل کنید که معرف یک مقدار عددی نیست . بالطبع نمی خواهید عمل بسط علامت انجام گیرد . این حالت هنگام کار با مقادیر براساس پیکسل (pixel) و گرافیک اغلب وجود دارد . در چنین شرایطی لازم است تا مقدار صفر در بیت بیش از حد مجاز قرار گیرد ، صرفنظر از اینکه مقدار قبلی در آن بیت چه بوده است . این حالت را انتقال فاقد علامت (usigned shift) می گویند. برای این منظور، از عملگر استفاده کنید که صفرها را در بیت بیش >>> حرکت به راست فاقد علامت در جاوا یعنی از حد مجاز منتقل می کند .

می باشد . در اینجا>>> کد قطعه ای زیر نشان دهنده عملگر a معادل 1- است که کلیـه 32 بیـت را در بـاینری روی 1 تنظـیم مـی

کند . این مقدار سپس 24 بیت بطرف راست انتقال می یابد، و 24 بیت بالایی را با صفرها پر می کند و بسط علامت معمولی را نادیده می گیرد . بدین ترتیب a معادل 255 می باشد; int a =- 1 .

a = a >>> 24;

اینجا همان عملیات را در شکل باینری مشاهده می کنید تا بهتر بفهمید چه تفاقی افتاده است : 1

در باینری بعنوان یک int 11111111 11111111 11111111 11111111 >>>24

255

در باینری بعنوان یک int´ 00000000 00000000 00000000 11111111 اغلب اوقات آنچنان سودمند که بنظر می رسد ، نبوده چون فقط برای >>> عملگر مقادیر 32 بیتی و 64 بیتی معنی دارد . بیاد آورید که مقادیر کوچکتر در عبارات بطور خودکار به int ارتقاءٔ می یابند . بدین ترتیب بسط علامت اتفاق افتاده و حرکت بجای مقادیر 8 بیتی و 16 بیتی روی مقادیر 32 بیتی انجام می شود . یعنی باید انتظار یک حرکت به راست فاقد علامت روی یک مقدار byte داشته باشیم که در بیت 7 ، صفر را قرار می دهد . اما واقعا" اینطور نیست ، چون در واقع مقدار 32 بیتی است که منتقل می شود . برنامه بعدی این تاثیری را نشان می دهد .

```
// Unsigned shifting a byte value.
class ByteUShift {
static public void main(String args[] ){
char hex[] = {
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
};
byte b =( byte )oxf1
byte c =( byte( )b >> 4);
byte d =( byte( )b >>> 4);
byte e =( byte( )b & oxff )>> 4);

System.out.println(" b = ox"+ hex[(b >> 4 )& oxof] + hex[b & oxof]);
System.out.println(" b >> 4 = ox" + hex[(c >> 4 )& oxof] + hex[c & oxof]);
System.out.println(" b >>> 4 = ox" + hex[(d >> 4 )& oxof] + hex[d & oxof]);
System.out.println("(b & oxof )>> 4 = ox" + hex[(e >> 4 )& oxof] + hex[e & oxof]);
}
}
```

چگونه هنگام کار با بایت ها عملی >>> خروجی این برنامه نشان میدهد که عملگر انجام نمی دهد . متغیر b بعنوان یک مقدار byteمنفی قراردادی در این نمایش تنظیم شده است . سپس مقدار byte در b ر که چهار مکان بطرف راست انتقال یافته به C

منتسب می شود که بخاطر بسط علامت مورد انتظار oxff است . سپس مقدارbyte درb ر که چهار مکان بطرف راست و فاقد علامت منتقل شده به d منتسب می شود که انتظار دارید oxof باشد ، اما در حقیقت oxff است چون بسط علامت هنگامیکه b بـه نوع int قبل از انتقال ارتقائ یافته اتفاق افتاده است . آخرین عبارت ، eرا در مقدارbyte متغیر b که با استفاده از عملگرAND با 8 بیت پوشانده شده تنظیم نموده و سپس چهار مکان بطرف راست منتقل می کند که مقدار مورد انتظار oxof را تولید می کند . دقت کنید که عملگر حرکت به راست فاقد علامت برای d استفاده نشد ، چون حالت بیت علامت بعد از AND شناخته شده است .

```
b=oxf1
b>>4=oxff
b>>>4=oxff
( b&oxff)>>4=oxof
```

انتسابهای عملگر رفتار بیتی

کلیه عملگرهای رفتار بیتی باینری یک شکل مختصر مشابه باعملگرهای جبری دارند که عمل انتساب را با عملیات رفتار بیتی ترکیب می کنند. بعنوان مثال ، دو دستور بعدی که مقدار a را چهار بیت به راست حرکت می دهند ، معادل یکدیگرند :

```
a = a >> 4;
a >>= 4;
```

بطور مشابه ، دو دستور زیر که a را به عبارت روش بیتی aoRb منتسب می کنند معادل یکدیگرند :

```
a = a | b;
a |= b;
```

برنامه بعدی تعدادی از متغیرهای عدد صحیح را بوجود آورده آنگاه از شکل مختصر انتسابهای عملگر رفتار بیتی برای کار کردن بااین متغیرها استفاده میکند :

```
class OpBitEquals {
 public static void main(String args[] ){
 int a = 1;
 int b = 2;
```

عملگر انتساب The Assignment Operator

عملگر انتساب علامت تکی تساوی = می باشد . عملگر انتساب در جاوا مشابه سایر زبانهای برنامه نویسی کار می کند . شکل کلی آن بصورت زیر است :

```
Var = expression;
```

عبارت متغیر

در اینجا نوع (var متغیر) باید با نوع (experssion عبارت) سازگار باشد. عملگر انتساب یک خصلت جالب دارد که ممکن است با آن آشنایی نداشته باشید : به شما امکان می دهد تا زنجیره ای از انتسابها بوجود آورید . بعنوان مثال ، این قطعه از یک برنامه را در نظر بگیرید :

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

این قطعه از برنامه مقدار 100 را با استفاده از یک دستور در متغیرهای x ، y، و z قرار می دهد . زیرا = عملگری است که مقدار عبارت سمت راست را جذب می کند . بنابراین مقدار Z=100 برابر 100 است که این مقدار به y منتسب شده و نیز به x منتسب خواهد شد . استفاده از " زنجیره ای از انتسابها " یک راه آسان برای قرار ادن یک مقدار مشترک در گروهی از متغیرهاست .

ارتقائ خودکار انواع در عبارات Automatic Type promotion in Expressions

علاوه بر انتسابها ، در شرایط دیگری هم تبدیلات خاص انواع ممکن است اتفاق بیفتد : در عبارات . حالتی را در نظر بگیرید که در یک عبارت ، میزان دقت لازم برای یک مقدار واسطه گاهی از دامنه هر یک از عملوندهایخود تجاوز می نماید . بعنوان مثال ، عبارت زیر را در نظر بگیرید :

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

نتیجه قلم واسطه a*b از دامنه هر یک از عملوندهای byte خود تجاوز می نماید. برای اداره این نوع مشکلات ، جاوا بطور خودکار هر یک از عملوندهای byte و short را هنگام ارزشیابی یک عبارت به int ارتقائ می دهد . این بدان معنی است که زیر عبارت

a*bبا استفاده از اعداد صحیح و نه byte اجرا می شود. بنابراین عدد 2000 نتیجه عبارت واسطه 50*40 مجاز است ، اگر چه a و bو هر دو بعنوان نوع byte مشخص شده اند .

همانقدر که ارتقائ خودکار مفیداست ، می تواند سبب بروز خطاهای زمان کامپایل (compile-time) گردد . بعنوان مثال ، این کد بظاهر صحیح یک مشکل را بوجود می آورد .

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

این کد تلاش می کند تا 50*2 را که یک مقدار کاملا" معتبر byte است به یک متغیر byte ذخیره کند . اما چون عملوندها بطور خودکار هنگام ارزشیابی عبارت بهint ارتقائ یافته اند ، جواب حاصله نیز به int ارتقائ یافته است . بنابراین جواب عبارت اکنون از نوع int است که بدون استفاده از تبدیل cast امکان نسبت دادن آن به یک byte وجود ندارد. این قضیه صادق است ، درست مثل همین حالت ، حتی اگر مقدار نسبت داده شده همچنان با نوع هدف سازگاری داشته باشد . در شرایطی که پیامدهای سر ریز (overflow) را درک می کنید، باید از یک تبدیل صریح cast نظیر مورد زیر استفاده نمایید .

```
byte b = 50;
b =( byte( )b * 2);
```

که مقدار صحیح عدد 100 را بدست می آورد .

قوانین ارتقائ انواع

علاوه بر ارتقائbyteوshort وبه int جاوا چندین قانون ارتقائانواع را تعریف کرده که قابل استفاده در عبارات می باشند . این قوانین بصورت زیر هستند . اول اینکه کلیه مقادیرbyte وshort وبه int ارتقائ می یابند ، همانگونه که قبلا" توضیح داده ایم . آنگاه اگر یک عملوند ، longباشد ، کل عبارت به long ارتقائ می یابد . اگر یک عملوند float باشد ، کل عبارت به float ارتقائ می یابد . اگر هر یک از عملوندها یک double باشند ، حاصل آنها double خواهد شد . برنامه بعدی نشان می دهد که چگونه هر یک از مقادیر در عبارت ارتقائ می یابد تا با آرگومان دوم به هر یک از عملگرهای دودویی ، مطابقت یابد .

```
class Promote {
public static void main(String args[] ){
byte b = 42;
char c = 'a';
short s = 1024;
int i = 50000;
```

```
float f = 5.67f;
double d =. 1234;
double result =( f * b )+( i / c( - )d * s);
System.out.println((f * b )+ " + " +( i / c )+ " - " +( d * s));
System.out.println("result = " + result);
}
}
```

اجازه دهید به ارتقائ انواع که در این خط از برنامه اتفاق افتاده ، دقیقتر

نگاه کنیم :

```
double result =( f * b )+( i / c(  )d * s);
```

در اولین زیر عبارت یعنیf*b ، b، به یک نوع float ارتقائ یافته و جواب زیر عبارت نیز از نوع float خواهد بود . در زیر عبارت بعدی یعنیi/c ، c، به یک نوعint ارتقائ یافته و جواب آن زیر عبارت نیز از نوع int خواهد بود . سپس در زیر عبارت d*s ، مقدار Sبه نوع double ارتقائ یافته و نوع زیر عبارت نیز double خواهد بود . در نهایت این سه مقدار واسطه ، float، int، double، در نظر گرفته می شوند . خروجی float بعلاوه int از نوع float خواهد شد . آنگاه این نتیجه منهای آخرین double به نوع double ارتقائ یافته ، که نوع مربوط به جواب نهایی

استفاده از بلوکهای کد Blocks of code

جاوا این امکان را فراهم نموده تا دو یا چند دستور در بلوکهای کد گرد آوری شوند که آنها را معمولا " code blocksمی نامند . اینکار با محصور کردن دستورات بین ابروهای باز و بسته انجام می گیرد .یکبار که یک بلوک کد ایجاد می شود، این بلوک که تبدیل به یک واحد منطقی شده و هر جایی که یک دستور ساده بتوان استفاده نمود، مورد استفاده قرار می گیرد .بعنوان مثال ، یک بلوک ممکن است هدف دستوراتif و یا for جاوا باشد . دستور if زیر را در نظر بگیرید :

```
if(x < y ){ // begin a block
x = y;
y = 0;
} // end of block
```

در اینجا اگر x کوچکتر از y باشد ، آنگاه هر دو دستور موجود در داخل بلوک اجرا خواهند شد . بنابراین دو دستور داخل بلوک تشکیل یک واحد منطقی داده اند و آنگاه اجرای یک دستور منوط به اجرای دستور دیگر خواهد بود . نکته کلیدی در اینجا این است

که هر گاه لازم باشد دو یا چند دستور را بطور منطقی پیوند دهید توسط ایجاد یک بلوک اینکار را انجام می دهید . به یک مثال دیگر نگاه کنید. برنامه بعدی از یک بلوک کد بعنوان هدف (target) یک حلقه for استفاده می کند .

```
/*
Demonstrate a block of code.
Call this file "BlockTest.java"
*/
class BlockTest {
public static void main(String args[] ){
int x/ y;
y = 20;
// the target of this loop is a block
for(x = 0; x<10; x++ ){
System.out.println("This is x :" + x);
System.out.println("This is y :" + y);
y = y - 2;
}
}
}
```

خروجی این برنامه بقرار زیر می باشد :

```
This is x:0
This is y:20
This is x:1
This is y:18
This is x:2
This is y:16
This is x:3
This is y:14
This is x:4
This is y:12
This is x:5
This is y:10
This is x:6
This is y:8
This is x:7
```

```
This is y:6
This is x:8
This is y:4
This is x:9
This is y:2
```

در این حالت ، هدف حلقه for یک بلوک کد است نه یک دستور منفرد. بدین ترتیب هر بار که حلقه تکرار می شود ، سه دستور داخل بلوک اجرا خواهد شد . این حقیقت در خروجی تولید شده توسط برنامه کاملا" هویداست . همانگونه که بعدا" خواهید دید، بلوکهای کد دارای ویژگیها و کاربردهای دیگری هم هستند . اما دلیل اصلی حضور آنها ایجاد واحدهای منطقی و تفکیک ناپذیر از باشد.

استفاده از پرانتزها

پرانتزها حق تقدم عملیاتی را که دربرگرفته اند ، افزایش می دهند . اینکار اغلب برای نگهداری نتیجه دلخواهتان ضروری است . بعنوان مثال ، عبارت زیر را در نظر بگیرید :

```
a >> b + 3
```

این عبارت ابتدا 3 را به b اضافه نموده و سپس a را مطابق آن نتیجه بطرف راست حرکت می دهد. این عبارت را می توان با استفاده از پرانتزهای اضافی بصورت زیر دوباره نویسی نمود :

```
a >>( b + 3)
```

اما ، اگر بخواهید ابتدا a را با مکانهای b بطرف راست حرکت داده و سپس 3 را به نتیجه آن اضافه کنید ، باید عبارت را بصورت زیر در پرانتز قرار دهید3 +(a >> b) :

علاوه بر تغییر حق تقدم عادی یک عملگر ، پرانتزها را می توان گاهی برای روشن نمودن مفهوم یک عبارت نیز بکار برد . برای هر کسی که کد شما را می خواند، درک یک عبارت پیچیده بسیار مشکل است . اضافه نمودن پرانتزهای اضافی و روشنگر به عبارات پیچیده می تواند از ابهامات بعدی جلوگیری نماید. بعنوان مثال ، کدامیک از عبارات زیر راحت تر خوانده و درک می شوند ?

```
a | 4 + c >> b & 7 || b > a % 3
( a |((( 4 + c )>> b )& 7 ))||( b >( a % 3))
```

یک نکته دیگر : پرانتزها (بطور کلی خواه اضافی باشند یا نه) سطح عملکرد برنامه شما را کاهش نمی دهند. بنابراین ، اضافه کردن پرانتزها برای کاهش ابهام نفی روی برنامه شما نخواهد داشت.

عملگر ?

جاوا شامل یک عملگر سه تایی ویژه است که می تواند جایگزین انواع مشخصی از دستورات if-then-else باشد. این عملگر علامت ? است و نحوه کار آن در جاوا مشابه با C و C++ و است . ابتدا کمی گیج کننده است ، اما می توان از ? براحتی و با کارایی استفاده نمود شکل کلی این عملگر بصورت زیر است :

experssion 1? experssion2 :experssion3

در اینجا experssion1 می تواند هر عبارتی باشد که با یک مقدار بولی سنجیده می شود . اگر experssion1 صحیح true باشد ، آنگاه experssion2 سنجیده می شود در غیر اینصورت experssion3 ارزیابی خواهد شد . نتیجه عملیات ? همان عبارت ارزیابی شده است . هر دو عبارتexperssion2 وexperssion3 و باید از یک نوع باشند که البته void نمی تواند باشد . در اینجا مثالی برای استفاده از عملگر ? مشاهده می کنید :

ratio = denom == 0 ? 0 : num / denom;

هنگامیکه جاوا این عبارت انتساب را ارزیابی می کند ، ابتدا به عبارتی که سمت چپ علامت سئوال قرار دارد ، نگاه می کند. اگر denomمساوی صفر باشد ، آنگاه عبارت بین علامت سئوال و علامت (colon) ارزیابی شده و بعنوان مقدار کل عبارت ? استفاده می شود . اگر denom مساوی صفر نباشد ، آنگاه عبارت بعد از (colon) ارزیابی شده و برای مقدار کل عبارت ? استفاده می شود . نتیجه تولید شده توسط عملگر ? سپس به ratio نسبت داده می شود . در زیر برنامه ای مشاهده می کنید که عملگر ? را نشان می دهد . این برنامه از عملگر فوق برای نگهداری مقدار مطلق یک متغیر استفاده می کند .

```
// Demonstrate ?.
class Ternary {
public static void main(String args[] ){
int i/ k;

i = 10;
k = i < 0 ?- i : i; // get absolute value of i
System.out.print("Absolute value of ");
System.out.println(i + " is " + k);
}
}
```

خروجی این برنامه بصورت زیر می باشد :

```
Absolute value of 10 is 10
```

```
Absolute value of- 10 is 10
```

حق تقدم عملگر

جدول زیر ترتیب حق تقدم عملگرهای جاوا را از بالاترین اولویت تا پایین ترین نشان می دهد . دقت کنید که در سطر اول اقلامی وجود دارد که معمولا" بعنوان عملگر درباره آنها فکر نمی کنید : پرانتزها ، کروشه ها و عملگر نقطه .

1. Highest
2. () [].
3. ++ -- ~ !
4. / %
5. +-
6. >> >>> <<
7. >= < <=
8. == !=
9. &
10. ^
11. |
12. &&
13. ||
14. ?:
15. = op=
16. Lowest

از پرانتزها برای تغییر حق تقدم یک عملیات استفاده می شود . قبلا" خوانده اید که کروشه های دوتایی نمایه سازی آرایه ها را فراهم می سازند . عملگرهای نقطه یاناستفاده شده که بعدا" مورد بررسی قرار خواهیم داد .

منابع :

http://www.irandevelopers.com/
http://docs.sun.com

نویسنده :
mamouri@ganjafzar.com محمد باقر معموری
ویراستار و نویسنده قسمت های تکمیلی :
zehs_sha@yahoo.com احسان شاه بختی

کتاب :

Java در 21 روز انتشارات نص
برنامه نویسی شی گرا انتشارات نص

معرفی کلاسها در جاوا

کلاس هسته اصلی جاوا است . کلاس یک ساختار منطقی است که تمامیت زبان جاوا بر آن استوار شده ، زیرا شکل (shape) و طبیعت یک شی ئ را روشن می کند . کلاس همچنین شکل دهنده اساس برنامه نویسی شی ئ گرا در جاوا می باشد .هر مفهومی که مایلید در یک برنامه جاوا پیاده سازی نمایید باید ابتدا داخل یک کلاس کپسول سازی شود . از این پس می آموزید چگونه یک کلاس را برای تولید اشیائ استفاده کنید. همچنین درباره روشها (methods) و سازنده ها (constructors) و واژه کلیدی this مطالبی می آموزید.

بنیادهای کلاس در جاوا

کلاسهای تولید شده در بحثهای گذشته فقط برای کپسول سازی روش main() استفاده می شد ، که برای نشان دادن اصول دستور زبان جاوا مناسب بودند . شاید بهترین چیزی که باید درباره یک کلاس بدانید این است که کلاس یک نوع جدید داده را تعریف می کند . هربار که این نوع تعریف شود ، می توان از آن برای ایجاد اشیائی از همان نوع استفاده نمود . بنابراین ، یک کلاس قالبی (template) برای یک شی ئ است و یک شی ئ نمونه ای (instance) از یک کلاس است . چون شی ئ یک نمونه از یک کلاس است غالبا" کلمات شی ئ (object)و نمونه (instance) را بصورت مترادف بکار می بریم .

شکل عمومی یک کلاس

هنگامیکه یک کلاس را تعریف می کنید ، در حقیقت شکل و طبیعت دقیق آن کلاس را اعلان می کنید . ابتکار را با توصیف داده های موجود در آن کلاس و کدهایی که روی آن داده ها عمل می کنند ، انجام می دهید . در حالیکه کلاسها ممکن است خیلی ساده فقط شامل داده یا فقط کد باشند، اکثر کلاسهای واقعی هردو موضوع را دربرمیگیرند .بعدا" خواهید دید که کد یک کلاس ، رابط آن به داده های همان کلاس را توصیف میکند .یک کلاس را با واژه کلیدی class اعلان می کنند . کلاسهایی که تا بحال استفاده شده اند ، نوع بسیار محدود از شکل کامل کلاسها بوده اند . خواهید دید که کلاسها می توانند (و معمولا" هم) بسیار پیچیده تر باشند . شکل عمومی توصیف یک کلاس به شرح زیر است :

```
type methodname2(parameter-list ){
// body of method
}
//...
type methodnameN(parameter-list ){
// body of method
}
}
```

داده یا متغیرهایی که داخل یک کلاس تعریف شده اند را متغیرهای نمونه (instance variables)می نامند . کدها ، داخل روشها (methods)قرار می گیرند . روشها و متغیرهای تعریف شده داخل یک کلاس را اعضائ (members) یک کلاس می نامند . در اکثر کلاسها ، متغیرهای نمونه یا روی روشهای تعریف شده برای آن کلاس عمل کرده یا توسط این روشها مورد دسترسی قرار می گیرند . بنابراین ، روشها تعیین کننده چگونگی استفاده از داده های یک کلاس هستند . متغیرهای تعریف شده داخل یک کلاس ، متغیرهای نمونه خوانده شده زیرا هر نمونه از کلاس (یعنی هر شی ئ یک کلاس) شامل کپی خاص خودش از این متغیرهاست . بنابراین داده مربوط به یک شی ئ ، جدا و منحصر بفرد از داده مربوط به شی ئ دیگری است . ما بزودی این نکته را بررسی خواهیم نمود ، اما فعلا" باید این نکته بسیار مهم را بیاد داشته باشید . کلیه روشها نظیر main()همان شکل عمومی را دارند که تاکنون استفاده کرده ایم . اما ، اکثر روشها را بعنوانstatic یا publicا توصیف نمی کنند . توجه داشته باشید که شکل عمومی یک کلاس ، یک روش main() را توصیف نمی کند . کلاسهای جاوا لزومی ندارد که یک روش main() داشته باشند . فقط اگر کلاس ، نقطه شروع برنامه شما باشد ، باید یک روش main() را توصیف نمایید . علاوه بر این ، ریز برنامه ها (applets) اصولا" نیازی به روش main() ندارند . نکته : برنامه نویسان C++ آگاه باشند که اعلان کلاس و پیاده سازی روشها در یک مکان ذخیره شده و بصورت جداگانه تعریف نمی شوند. این حالت گاهی فایلهای خیلی بزرگ java ایجاد می کند ، زیرا هر کلاس باید کاملا" در یک فایل منبع تکی تعریف شود . این طرح در جاوا رعایت شد زیرا احساس می شد که در بلند مدت ، در اختیار داشتن مشخصات ، اعلانها و پیاده سازی در یک مکان ، امکان دسترسی آسانتر کد را بوجود می آورد . یک کلاس ساده بررسی خود را با یک نمونه ساده از کلاسها شروع می کنیم . در اینجا کلاسی تحت غنوان Box وجود دارد که سه متغیر نمونه را تعریف می کند: width ، height، وdepth و فعلا "، کلاس Box دربرگیرنده روشها نیست .

```
class Box {
  double width;
  double height;
  double depth;
}
```

قبلا" هم گفتیم که یک کلاس نوع جدیدی از داده را توصیف می کند . در این مثال نوع جدید داده را Box نامیده ایم . از این نام برای اعلان اشیائ از نوع Box استفاده می کنید . نکته مهم این است که اعلان یک کلاس فقط یک الگو یا قالب را ایجاد می کند ، اما یک شی ئ واقعی بوجود نمی آورد . بنابراین ، کد قبلی ، شیئی از نوع Box را بوجود نمی آورد . برای اینکه واقعا" یک شی ئ Box را بوجود آورید ، باید از دستوری نظیر مورد زیر استفاده نمایید :

```
Box mybox = new Box(); // create a Box object called mybox
```

پس از اجرای این دستور ، myboxنمونه ای از Box خواهد بود .و بدین ترتیب این شی ئ وجود فیزیکی و واقعی پیدا می کند . مجددا" بیاد داشته باشید که هر بار یک نمونه از کلاسی ایجاد می کنید ، شیئی ایجاد کرده اید که دربرگیرنده کپی (نسخه خاص) خود از هر متغیر نمونه تعریف شده توسط کلاس خواهد بود . بدین ترتیب ، هر شی ئ Box دربرگیرنده کپی های خود از متغیرهای نمونهwidth ،

heidhtو depth و می باشد . برای دسترسی به این متغیرها از عملگر نقطه (.) استفاده می کنید . عملگر نقطه ای ، نام یک شی ئ را با نام یک متغیر نمونه پیوند می دهد . بعنوان مثال ، برای منتسب کردن مقدار 100 به متغیر width در mybox ر ، از دستور زیر استفاده نمایید :

```
mybox.width = 100;
```

این دستور به کامپایلر می گوید که کپی width که داخل شی ئ mybox قرار گرفته را معادل عدد 100 قرار دهد . بطور کلی ، از عملگر نقطه ای برای دسترسی هم به متغیرهای نمونه و هم به روشهای موجود در یک شی ئ استفاده می شود . دراینجا یک برنامه کامل را مشاهده میکنید که از کلاس Box استفاده کرده است :

```
/* A program that uses the Box class.
Call this file BoxDemo.java
/*
class Box {
double width;
double height;
double depth;
}
```

```
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[] ){
Box mybox = new Box();
double vol;

// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;

// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;

System.out.println("Volume is " + vol);
```

```
}
}
```

فایلی را که دربرگیرنده این برنامه است باید با نام BoxDemo.java بخوانید زیرا روش main() در کلاس BoxDemo و نه در کلاس Box قرار گرفته است . هنگامیکه این برنامه را کامپایل می کنید ، می بینید که دو فایل .class ایجاد شده اند ، یکی برای Box و دیگری برای BoxDemo . کامپایلر جاوا بطور خودکار هر کلاس را در فایل .class مربوط به خودش قرار می دهد . ضرورتی ندارد که کلاس Box و BoxDemo و هر دو در یک فایل منبع قرار گیرند . می توانید هر کلاس را در فایل خاص خودش گذاشته و آنها را بترتیب Box.java و BoxDemo.java و بنامید . برای اجرای این برنامه باید BoxDemo.class را اجرا کنید . پس از اینکار حاصل زیر را بدست می آورید :

```
Volume is 3000
```

قبلا" هم گفتیم که هر شی ئ دارای کپی های خاص خودش از متغیرهای نمونه است . یعنی اگر دو شی ئ Box داشته باشید، هرکدام بتنهایی کپی (یا نسخه ای) از lenght width و height و خواهند داشت . مهم است بدانید که تغییرات در متغیرهای نمونه یک شی ئ تاثیری روی متغیرهای نمونه کلاس دیگر نخواهد داشت . بعنوان مثال ، برنامه بعدی دو شی ئ Box را اعلان می کند :

```
// This program declares two Box objects.

class Box {
   double width;
   double height;
   double depth;
}

class BoxDemo2 {
  public static void main(String args[] ){

Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;

// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

```
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;

// compute volume of first box
vol = mybox1.width * mybox1.height * mybox1.depth;
System.out.println("Volume is " + vol);

// compute volume of second box
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Volume is " + vol);
}
}
```

خروجی تولید شده توسط این برنامه بقرار زیر می باشد :

```
Volume is 3000
Volume is 162
```

تعریف کنترل دسترسی

می دانید که کپسول سازی ، داده ها را با کدی که با آن داده ها سر و کار دارد پیوند میدهد. اما کپسول سازی یک حصلت بسیار مهم دیگر هم دارد: کنترل دستیابی . (access control) . از طریق کپسول سازی ، می توانید کنترل کنید چه بخشهایی از یک برنامه می توانند به اعضائ یک کلاس دسترسی داشته باشند . با کنترل نمودن دستیابی ، می توانید از سوئ استفاده جلوگیری نمایید .بعنوان مثال ، اجازه دادن برای دسترسی به داده ها فقط از طریق یک مجموعه خوش تعریف از روشها ، مانع سوئ استفاده از آن داده ها می شود . بنابراین ، وقتی این کار بخوبی پیاده سازی شود یک کلاس یک " جعبه سیاه (black box()" تولید می کند که امکان دارد مورد استفاده قرار گیرد، اما عملکرد درونی آن جعبه در معرض دسترسی دیگران قرارنخواهد داشت . اما کلاسهایی که قبلا" تعریف شده اند این هدف را بطور کامل برآورده نمی سازند . بعنوان مثال ، کلاس stack را درنظر بگیرید .اگرچه این امر حقیقت دارد که روشهای push() و pop()رابطهای کنترل شده ای به پشته هستند ، اما این رابط تاکید نشده

است . یعنی این امکان وجود دارد که بخش دیگری از برنامه این روشها را دور زده و مستقیما" به پشته دسترسی یابد . البته این خاصیت در دستان افراد ناجور، سبب مشکلاتی خواهد شد . در این قسمت مکانیسمی به شما معرفی می کنیم که توسط آن با دقت تمام دسترسی به اعضائ مختلف یک کلاس را کنترل می کنید .

چگونگی دسترسی به یک عضو توسط " توصیفگر دسترسی (access specifier()" که اعلان آن عضو را تغییر میدهد، تعریف خواهد شد. جاوا مجموعه غنی از " توصیفگرهای دسترسی " را عرضه می کند . برخی جوانب کنترل دسترسی بشدت با وراثت و بسته ها (packages)مرتبط اند . (یک بسته ضرورتا" یک نوع گروه بندی ازکلاسها است (. اگر یکبار بنیادهای کنترل دسترسی را فرا گیرید ، آنگاه بقیه مطالب را براحتی درک خواهید نمود . توصیفگرهای دسترسی در جاواpublic، private وprotected و هستند. جاوا همچنین یک سطح دسترسی پیش فرض را تعریف کرده است protected . زمانی استفاده می شود که وراثت وجود داشته باشد .

با تعریف public وprivate و شروع می کنیم . اگر یک عضو کلاسی را با توصیفگرpublic توصیف می کنیم ، آن عضو توسط هر کد دیگری در برنامه قابل دسترسی خواهد بود . اگر یک عضو کلاسی را بعنوان private مشخص می کنیم ، پس آن عضو فقط توسط سایر اعضائ همان کلاس قابل دسترسی است . اکنون می فهمید که چرا همیشه قبل از main() مشخصگر public قرار می گرفت . این روش توسط کدی خارج از برنامه یعنی توسط سیستم حین اجرای جاوا فراخوانی خواهد شد . اگر هیچ توصیفگر دسترسی استفاده نشده باشد ، آنگاه بصورت پیش فرض عضو یک کلاس ، داخل بسته مربوط به خودpublic است ، اما خارج از بسته مربوط به خود قابل دسترسی نمی باشد .

در کلاسهایی که تاکنون توسعه داده ایم ، کلیه اعضائ یک کلاس از حالت دسترسی پیش فرض که ضرورتا" همان public است ، استفاده کرده اند . اما همیشه این حالت مطلوب شما نیست . معمولا" مایلید تا دسترسی به اعضائ داده ای یک کلاس را محدود نمایید و فقط از طریق برخی روشها امکان پذیر سازید . همچنین ، شرایطی وجود دارند که مایلید روشهایی را که برای یک کلاس اختصاصی هستند ، تعریف نمایید .

یک توصیفگر دسترسی قبل از سایر مشخصات نوع عضو یک کلاس قرار می گیرد . یعنی که این توصیفگر باید شروع کننده دستور اعلان یک عضو باشد ، یک مثال را مشاهده می کنید :

```
public int i;
private bouble j;
private int myMethod(int a/ char b ){ //...
```

برای درک تاثیرات دسترسی عمومی (public) و اختصاصی (private) برنامه بعدی را در نظر بگیرید :

```
/* This program demonstrates the difference between
 public and private.
 */
class Test {
int a; // default access
public int b; // public access
private int c; // private access

// methods to access c
void setc(int i ){ // set c's value
c = i;
}
int getc (){ // get c's value
return c;
}
}

class AccessTest {
public static void main(String args[] ){
Test ob = new Test();
```

```
// These are OK/ a and b may be accessed directly
ob.a = 10;
ob.b = 20;

// This is not OK and will cause an error
// ob.c = 100; // Error!

// You must access c through its methods
ob.setc(100); // OK

System.out.println("a/ b/ and c :" + ob.a + " " +
ob.b + " " + ob.getc());
}
}
```

همانطوریکه مشاهده می کنید ، داخل کلاس Test ، a، از دسترسی پیش فرض استفاده می کند که دراین مثال مطابق مشخص نمودن publicاست b. بطور صریح بعنوان public مشخص شده است c . دارای دسترسی اختصاصی است . بدان معنی که این عضو توسط کدهای خارج از کلاس قابل دسترسی نخواهد بود. همچنین در داخل کلاس Access Test ، نمیتوانc را بصورت مستقیم مورد استفاده قرار داد . این عضو را باید از طریق روش های عمومی اش یعنی setc() و getc()مورد دسترسی قرار داد. اگر نشانه صحیح (comment) را ابتدای خط تغییر مکان دهید :

+ // ob.c = 100; // Error!

آنگاه نمی توانید این برنامه را کامپایل کنید که علت آن هم نقض دسترسی (access violation) است . برای اینکه با جنبه های عملی کنترل دسترسی در مثالهای عملی تر آشنا شوید روایت توسعه یافته از کلاس stack را در برنامه زیر مشاهده فرمایید :

```
// This class defines an integer stack that can hold 10 values.
class Stack {
/* Now/ both stck and tos are private .This means
that they cannot be accidentally or maliciously
altered in a way that would be harmful to the stack.
*/
private int stck[] = new int[10];
private int tos;

// Initialize top-of-stack
Stack (){
```

```
tos =- 1;
}
// Push an item onto the stack
void push(int item ){
if(tos==9)
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item frome the stack
int pop (){
if(tos < 0 ){
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
```

همانطوریکه مشاهده می کنید ، هم stck که پشته را نگهداری می کند و هم tos که نمایه بالای پشته است بعنوان private مشخص شده اند . این بدان معنی است که آنها قابل دسترسی و جایگزینی جزئ از طریق push() و pop() نیستند . بعنوان مثال اختصاصی نمودن tos سایر بخشهای برنامه اتان را در مقابل قرار دادن غیر عمدی مقداری که فراتر از انتهای آرایه stck باشد ، محافظت می کند . برنامه بعدی نشاندهنده کلاس توسعه یافته stack است .سعی کنید خطوط غیرتوضیحی را حرکت دهید تا بخود اثبات کنید که اعضای stck و tos در حقیقت غیر قابل دسترسی هستند .

```
class TestStack {
public static void main(String args[] ){
Stack mystack1 = new Stack();
Stack mystack2 = new Stack();

// push some numbers onto the stack
for(int i=0; i<10; i++ )mystack1.push(i);
for(int i=10; i<20; i++ )mystack2.push(i);

// pop those numbers off the stack
```

```
System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<10; i++)
System.out.println(mystack2.pop());
// these statements are not legal
// mystack1.tos =- 2;
// mystack2.stck[3] = 100;
}
}
```

اگرچه روشها معمولا" دسترسی به داده های تعریف شده توسط یک کلاس را کنترل می کنند ، اما همیشه هم اینطور نیست . کاملا" بجاست که هرگاه که دلیل خوبی برای اینکار داشته باشیم ، اجازه دهیم تا یک متغیر نمونه publicباشد . بعنوان مثال اکثر کلاسهای ساده با کمترین توجه نسبت به کنترل دسترسی به متغیرهای نمونه ایجاد شده اند و این بی توجهی فقط بلحاظ حفظ سادگی مثال بوده است.

محافظت دسترسی Access protection

قبلا" می دانستید که دسترسی به یک عضو private در یک کلاس فقط به سایر اعضائ همان کلاس واگذار شده است . بسته ها بعد دیگری به کنترل دسترسی می افزایند . همانطوریکه خواهید دید ، جاوا سطوح چندی از محافظت برای اجازه کنترل خوب طبقه بندی شده روی رویت پذیری متغیرها و روشهای داخل کلاسها ، زیر کلاسها و بسته ها فراهم می نماید . کلاسها و بسته ها هر دو وسایلی برای کپسول سازی بوده و دربرگیرنده فضای نام

و قلمرو متغیرها و روشها می باشند . بسته ها بعنوان ظروفی برای کلاسها و سایر بسته های تابعه هستند . کلاسها بعنوان ظروفی برای داده ها و کدها می باشند . کلاس کوچکترین واحد مجرد در جاوا است . بلحاظ نقش متقابل بین کلاسها و بسته ها ، جاوا چهار طبقه بندی برای رویت پذیری اعضائ کلاس مشخص کرده است :

1. زیر کلاسها در همان بسته
2. غیر زیر کلاسها در همان بسته
3. زیر کلاسها در بسته های مختلف
4. کلاسهایی که نه در همان بسته و نه در زیر کلاسها هستند .

سه مشخصگر دسترسی یعنی private ، public و protected و فراهم کننده طیف گوناگونی از شیوه های تولید سطوح چند گانه دسترسی مورد نیاز این طبقه بندیها هستند . جدول زیر این ارتباطات را یکجا نشان داده است .

private Nomodifier protected public |

همان کلاس Yes Yes Yes Yes

همان بسته زیر کلاس Yes Yes Yes No

همان بسته غیر زیر کلاس Yes Yes Yes No

بسته های مختلف زیر کلاس Yes Yes No No

بسته های مختلف غیر زیر کلاس Yes No No No

اگرچه مکانیسم کنترل دسترسی در جاوا ممکن است بنظر پیچیده باشد، اما میتوان آن را بصورت بعدی ساده گویی نمود . هر چیزی که بعنوان public اعلان شود از هر جایی قابل دسترسی است . هر چیزی که بعنوان private اعلان شود خارج از کلاس خودش قابل رویت نیست . وقتی یک عضو فاقد مشخصات دسترسی صریح و روشن باشد ، آن عضو برای زیر کلاسها و سایر کلاسهای موجود در همان بسته قابل رویت است . این دسترسی پیش فرض است . اگر می خواهید یک عضو ، خارج از بسته جاری و فقط به کلاسهایی که مستقیما" از کلاس شما بصورت زیر کلاس درآمده اند قابل رویت باشد ، پس آن عضو را بعنوان protected اعلان نمایید . یک کلاس فقط دو سطح دسترسی ممکن دارد : پیش فرض و عمومی . (public) وقتی یک کلاس بعنوان public اعلان می شود ، توسط هر کد دیگری قابل دسترسی است . اگر یک کلاس دسترسی پیش فرض داشته باشد ، فقط توسط سایر کدهای داخل همان بسته قابل دسترسی خواهد بود .

یک مثال از دسترسی

مثال بعدی کلیه ترکیبات مربوط به اصلاحگرهای کنترل دسترسی را نشان می دهد . این مثال دارای دو بسته و پنج کلاس است . بیاد داشته باشید که کلاسهای مربوط به دو بسته متفاوت ، لازم است در دایرکتوریهایی که بعداز بسته مربوطه اشان نام برده شده در این مثال p1 و p2 و ذخیره می شوند . منبع اولیه بسته سه کلاس تعریف می کند protection : ، Derived و samepackage و . اولین کلاس چهار متغیر int را در هر یک از حالات مختلف مجاز تعریف می کند. متغیر n با حفاظت پیش فرض اعلان شده است m-pri . بعنوان private ، n-pro بعنوان protected و n-pub و بعنوان public می باشند .

هر کلاس بعدی در این مثال سعی می کند به متغیرهایی در یک نمونه از یک کلاس دسترسی پیدا کند . خطوطی که بلحاظ محدودیتهای دسترسی ، کامپایل نمی شوند با استفاده از توضیح یک خطی // از توضیح خارج شده اند . قبل از هر یک از این خطوط توضیحی قرار دارد که مکانهایی را که از آنجا این سطح از حفاظت اجازه دسترسی می یابد را فهرست می نماید .

دومین کلاس Derived یک زیر کلاس از protection در همان بسته p1 است . این مثال دسترسی Derived را به متغیری در protectionبرقرار می سازد بجز n-pri که یک private است .

سومین کلاس Samepackage یک زیر کلاس از protection نیست ، اما در همان بسته قرار دارد و بنابراین به کلیه متغیرها بجز n-priدسترسی خواهد داشت .

```
package p1;

public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;

public Protection (){
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}

class Derived extends Protection {
Derived (){
System.out.println("derived constructor");
System.out.println("n = " + n);

// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
```

```
System.out.println("n_pub = " + n_pub);
}
}

class SamePackage {
SamePackage (){
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);

// class only
// System.out.println("n_pri = " + p.n_pri);

System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

اکنون کد منبع یک بسته دیگر یعنی p2 را مشاهده می کنید . دو کلاس تعریف شده در p2 دو شرایطی را که توسط کنترل دسترسی تحت تاثیر قرار گرفته اند را پوشش داده است . اولین کلاس یعنی protection 2 یک زیر کلاس p1.protection است . این کلاس دسترسی به کلیه متغیرهای مربوط به p1.protection را بدست می آورد غیر از n-pri(چون private است) و n_ متغیری که با محافظت پیش فرض اعلان شده است . بیاد داشته باشید که پیش فرض فقط اجازه دسترسی از داخل کلاس یا بسته را می دهد نه از زیر کلاس های بسته های اضافی . در نهایت ، کلاس otherpackage فقط به یک متغیر n-pub که بعنوان public اعلان شده بود دسترسی خواهد داشت .

```
package p2;

class Protection2 extends p1.Protection {
Protection2 (){
System.out.println("derived other package constructor");

// class or package only
System.out.println("n = " + n);

// class only
// System.out.println("n_pri = " + n_pri);
```

```
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}

class OtherPackage {
OtherPackage (){
p1.Protection p = new p1.protection();
System.out.println("other package contryctor");

// class or package only
System.out.println("n = " + p.n);

// class only
// System.out.println("n_pri = " + p.n_pri);

// class/ subclass or package only
// System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

اگر مایلید تا این دو بسته را آزمایش کنید ، در اینجا دو فایل آزمایشی وجود دارد که می توانید از آنها استفاده نمایید. یکی از فایلها برای بسته p1 را در زیر نشان داده ایم :

```
// Demo package p1.
package p1;

// Instantiate the various classes in p1.
public class Demo {
public static void main(String args[] ){
Protection ob1 = new Protection();
Derived ob2 = new Drived();
SamePackage ob3 = new SamePackage();
}
}
```

فایل آزمایشی برای p2 بقرار زیر می باشد :

+ // Demo package p2.

+ package p2;

+ // Instantiate the various classes in p2.

+ public class Demo {

+ public static void main(String args[]){

+ Protection2 ob1 = new

سازندگان Constructors

خیلی خسته کننده است که هربار یک نمونه ایجاد می شود کلیه متغیرهای یک کلاس را مقداردهی اولیه نماییم . حتی هنگامیکه توابع سهل الاستفاده نظیر setDim() را اضافه می کنید ، بسیار ساده تر و دقیق تر آن است که کلیه تنظیمات در زمان ایجاد اولیه شی ئ انجام شود . چون نیاز به مقدار دهی اولیه بسیار رایج است ، جاوا به اشیائ امکان می دهد تا در زمان ایجاد شدن خودشان را مقدار دهی اولیه نمایند . این مقدار دهی اولیه خودکار بااستفاده از سازنده (constructor) انجام می گیرد .یک سازنده بمحض ایجاد یک شی ئ بلافاصله آن را مقدار دهی اولیه می نماید . این سازنده نام همان کلاسی را که در آن قرار گرفته اختیار نموده و از نظر صرف و نحو مشابه یک روش است . وقتی یکبار سازنده ای را تعریف نمایید، بطور خودکار بلافاصله پس از ایجاد یک شی ئ و قبل از اینکه عملگر new تکمیل شود ، فراخوانی خواهد شد . سازندگان کمی بنظر عجیب می آیند زیرا فاقد نوع برگشتی و حتی فاقد void هستند . بخاطر اینکه نوع مجازی برگشتی سازنده یک کلاس ، همان نوع خود کلاس می باشد . این وظیفه سازنده است که وضعیت داخلی یک شی ئ را بگونه ای مقدار دهی اولیه نماید که کدی که یک نمونه را ایجاد می کند ، بلافاصله یک شی ئ کاملا" قابل استفاده و مقدار دهی شده بوجود آورد . می توانید مثال مربوط به Box را طوری بازنویسی کنید که ابعاد یک box وقتی که یک شی ئ ساخته می شود ، بطور خودکار مقدار دهی اولیه شوند . برای انجام این کار ، یک سازنده را جایگزین setDim() نمایید . کار را با تعریف یک سازنده ساده شروع می کنیم که بسادگی ابعاد هر box را بهمان مقادیر تنظیم میکند. این روایت جدید بقرار زیر است :

```
/* Here/ Box uses a constructor to initialize the
dimensions of a box.
*/
class Box {
double width;
double height;
double depth;

// This is the constructor for Box.
Box (){

System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}

// compute and return volume
double volume (){
```

```
return width * height * depth;
}
}

class BoxDemo6 {
public static void main(String args[] ){
// declare/ allocate/ and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();

System.out.println("volume is " + vol);

// get volume of second box
vol = mybox2.volume();
System.out.println("volume is " + vol);
}
}
```

پس از اجرای این برنامه ، خروجی آن بصورت زیر خواهد شد :

```
Constructing Box
Constructing Box
Volume is 1000
Volume is 1000
```

همانطوریکه مشاهده می کنید ، mybox1وmybox2 و هنگام ایجاد شدن بوسیله Box() مقداردهی اولیه میشوند. چون سازنده به همه boxها همان ابعاد یعنی 10 x10x10را می دهد ، هم mybox1 و هم mybox2 فضای اشغالی یکسانی خواهند داشت . دستور println()داخل Box() فقط بمنظور توصیف بهتر قرار گرفته است . اکثر توابع سازنده چیزی را نمایش نمی دهند. آنها فقط خیلی ساده ، اشیائ را مقدار دهی اولیه می کنند . قبل از ادامه بحث ، عملگر new را مجددا" بررسی میکنیم . همانطوریکه می دانید هنگامیکه یک شی ئ را اختصاص می دهید ، شکل عمومی زیر را مشاهده می کنید :

```
class-var = new classname();
```

اکنون می توانید بفهمید چرا پرانتزها بعد از نام کلاس مورد نیازند . آنچه واقعا" اتفاق می افتد این است که سازنده کلاس فراخوانی شده است . بدین ترتیب در خط

+ Box mybox1 = new Box();

newBox()سازنده Box() را فراخوانی می کند اگر یک سازنده برای کلاس تعریف نشود ، آنگاه جاوا یـک سـازنده پـیش فـرض بـرای کلاس ایجاد می کند . بهمین دلیل خط قبلی از کد در روایتهای اولیه Box که هنوز یک سازنده تعریف نشده بود ، کار می کرد. سازنده پیش فرض بطور خودکار کلیه متغیرهای نمونه را با عدد صفر، مقدار دهی اولیه می نماید . سازنده پیش فرض غالبا" برای کلاسهای ساده کفایت می کند اما برای کلاسهای پیچیده تر کفایت نمی کند . هر بار که سازنده خاص خود را تعریف کنید ، دیگر سازنده پـیش فـرض اسـتفاده نخواهـد شد .

سازندگان پارامتردار شده (parameterized)

هنگامیکه سازنده Box() در مثال قبلی یک شی ئ Box را مقدار دهی اولیه می کند چنـدان سـودمند نیسـت ، زیـرا کلیـه box هـا ، ابعاد یکسانی دارند . آنچه مورد نیاز است ، روشی برای ساخت اشیائ Box دارای ابعاد گوناگون است . ساده ترین راه افزودن پارامترها به سازنده است . همانطوریکه احتمالا" حدس می زنید ، این عمل سودمندی سازندگان را افزایش مـی دهـد . بعنـوان مثـال ، روایـت بعـدی Box یـک سازنده پارامتردار شده را تعریف می کند که ابعاد یک box را همانطوریکه آن پارامترها مشخص شده اند ، تنظیم مـی کنـد. دقـت کـافی بـه چگونگی ایجاد اشیائ Box داشته باشید .

```
/* Here/ Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;

// This is the constructor for Box.
Box(double w/ double h/ double d ){
width = w;
height = h;
depth = d;
}

// compute and return volume
double volume (){
return width * height * depth;
}
```

```
}

class BoxDemo7 {
public static void main(String args[] ){
// declare/ allocate/ and initialize Box objects
Box mybox1 = new Box(10/ 20/ 15);
Box mybox2 = new Box(3/ 6/ 9);

double vol;

// get volume of first box
vol = mybox1.volume();
System.out.println("volume is " + vol);

// get volume of second box
vol = mybox2.volume();
System.out.println("volume is " + vol);
}
}
```

خروجی این برنامه بقرار زیر است :

```
Volume is 3000
Volume is 162
```

همانطوریکه می بینید ، هر شی ئ همانطوریکه در پارامترهای سازنده اش مشخص شده مقدار دهی اولیه خواهد شد .

```
Box mybox1 = new Box(10, 20, 15);
```

هنگامیکه new شی ئ را ایجاد می کند ، مقادیر 10 ، 20 ، 15 به سازنده Box() گذر می کنند

وراثت inheritance

وراثت را یکی از سنگ بناهای برنامه نویسی شي ئ گراست ، زيرا امکان ایجاد طبقه بندیهای سلسله مراتبی را بوجود می آورد . با استفاده از وراثت ، می توانید یک کلاس عمومی بسازید که ویژگیهای مشترک یک مجموعه اقلام بهم مرتبط را تعریف نماید . این کلاس بعدا" ممکن است توسط سایر کلاسها بارث برده شده و هر کلاس ارث برنده چیزهایی را که منحصر بفرد خودش باشد به آن اضافه نماید . در روش شناسی جاوا ، کلاسی که بارث برده می شود را کلاس بالا (superclass) می نامند . کلاسی که عمل ارث بری را انجام داده و ارث برده است را زیر کلاس (subclass) می نامند . بنابراین ، یک " زیر کلاس " روایت تخصصی تر و مشخص تر از یک " کلاس بالا " است . زیر کلاس ، کلیه متغیرهای نمونه و روشهای توصیف شده توسط کلاس بالا را بارث برده و منحصر بفرد خود را نیز اضافه می کند .

مبانی وراثت

برای ارث بردن از یک کلاس ، خیلی ساده کافیست تعریف یک کلاس را با استفاده از واژه کلیدی extends در کلاس دیگری قرار دهید . برای فهم کامل این مطلب ، مثال ساده ای را نشان می دهیم . برنامه بعدی یک کلاس بالا تحت نام A و یک زیر کلاس موسوم به B ایجاد می کند . دقت کنید که چگونه از واژه کلیدی extends استفاده شده تا یک زیر کلاس از A ایجاد شود .

```
// A simple example of inheritance.

// Create a superclass.
class A {
int i, j;

void showij (){
System.out.println("i and j :" + i + " " + j);
}
}

// Create a subclass by extending class A.
class B extends A {
int k;
void showk (){
```

```
System.out.println("k :" + k);
}
void sum (){
System.out.println("j+j+k :" +( i+j+k));
}
}

class SimpleInheritance {
public static void main(String args[] ){
A superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20;
System.out.println("Contents of superOb :");
superOb.showij();
System.out.println();

/* The subclass has access to all public members of
its superclass .*/
subOb.i = 7;
subOb.j = 8;
subOb.k = 9;
System.out.println("Contents of subOb :");
subOb.showj();
subOb.showk();
System.out.println();

System.out.println("Sum of i/ j and k in subOb:");
subOb.sum();
}
}
```

خروجی این برنامه ، بقرار زیر می باشد :

```
Contents of superOb:
i and j :10 20

Contents of subOb:
```

```
i and j :7 8
k :9

Sum of i/ j and k in subOb:
i+j+k :24
```

همانطوریکه می بینید ، زیر کلاس B دربرگیرنده کلیه اعضائ کلاس بالای مربوطه یعنی A است . بهمین دلیل است که subob می تواند به iو j و دسترسی داشته و showij() را فراخوانی نماید . همچنین داخل sum() می توان بطور مستقیم i و j و همانگونه که قبلا" بخشی از B بودند ، ارجاع نمود . اگرچه A کلاس بالای B می باشد ، اما همچنان یک کلاس کاملا" مستقل و متکی بخود است . کلاس بالا بودن برای یک زیر کلاس بدان معنی نیست که نمی توان خود آن کلاس بالا را بتنهایی مورد استفاده قرار داد . بعلاوه ، یک زیر کلاس می تواند کلاس بالای یک زیر کلاس دیگر باشد . شکل عمومی اعلان یک class که از یک کلاس بالا ارث می برد ، بصورت زیر است :

```
class subclass-name extends superclass-name {
// body of class
}
```

برای هر زیر کلاسی که ایجاد می کنید ، فقط یک کلاس بالا می توانید تعریف کنید .جاوا از انتقال وراثت چندین کلاس بالا به یک کلاس منفرد پشتیبانی نمی کند . (از این نظر جاوا با C++ متفاوت است که در آن وراثت چند کلاسه امکان پذیر است .) قبلا" گفتیم که می توانید یک سلسله مراتب از وراثت ایجاد کنید که در آن یک زیر

کلاس ، کلاس بالای یک زیر کلاس دیگر باشد . اما ، هیچ کلاسی نمی تواند کلاس بالای خودش باشد.

دسترسی به اعضائ و وراثت

اگرچه یک زیر کلاس دربرگیرنده کلیه اعضائکلاس بالای خود می باشد، اما نمیتواند به اعضایی از کلاس بالا که بعنوان private اعلان شده اند ، دسترسی داشته باشد . بعنوان مثال ، سلسله مراتب ساده کلاس زیر را در نظر بگیرید :

```
/* In a class hierarchy/ private members remain
private to their class.
This program contains an error and will not
compile.
*/

// Create a superclass.
class A {
int i; // public by default
```

```
private int j; // private to A
void setij(int x/ int y ){
i = x;
j = y;
}
}

// A's j is not accessible here.
class B extends A {
int total;
void sum (){
total = i + j; // ERROR/ j is not accessible here
}
}

class Access {
public static void main(String args[] ){
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
}
}
```

این برنامه کامپایل نخواهد شد زیرا ارجاع به j داخل روش sum() درB سبب نقض دسترسی خواهد شد. از آنجاییکه j بعنوان private اعلان شده ، فقط توسط سایر اعضائ کلاس خودش قابل دسترسی است و زیر کلاسها هیچگونه دسترسی به آن ندارند . یادآوری : یک عضو کلاس که بعنوان private اعلان شده برای کلاس خودش اختصاصی خواهد بود . این عضو برای کدهای خارج از کلاسش از جمله زیر کلاسها ، قابل دسترسی نخواهد بود

.

یک مثال عملی تر

اجازه دهید به یک مثال عملی تر بپردازیم که قدرت واقعی وراثت را نشان خواهد داد . در اینجا ، روایت نهایی کلاس Box بنحوی گسترش یافته تا یک عنصر چهارم تحت نام weight را دربرگیرد . بدین ترتیب ، کلاس جدید شامل width ، height، depthو weightویک box خواهد بود .

```
// This program uses inheritance to extend Box.
class Box {
double width;
double height;
double depth;

// construct clone of an object
Box(Box ob ){ // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}

// constructor used when all dimensions specified
Box(double w/ double h/ double d ){
width = w;
height = h;
depth = d;
}

// constructor used when all dimensions specified
Box (){
width =- 1; // use- 1 to indicate
height =- 1; // an uninitialized
depth =- 1; // box
}

// compute and return volume
double volume (){
return width * height * depth;
}
}

// Here/ Box is extended to include weight.
class BoxWeight extends Box {
double weight; // weight of box

// constructor for BoxWeight
```

```
BoxWeight(double w/ double h/ double d/ double m ){
width = w;
height = h;
depth = d;
weight = m;
}
}

class DemoBoxWeight {
public static void main(String args[] ){
Boxweight mybox1 = new BoxWeight(10/ 20/ 15/ 34.3);
Boxweight mybox2 = new BoxWeight(2/ 3/ 4/ 0.076);
double vol;

vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();

vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
}
}
```

خروجی این برنامه بصورت زیر می باشد :

```
Volume of mybox1 is 3000
Weight of mybox1 is 34.3

Volume of mybox2 is 24
Weight of mybox2 is 0.076
Boxweight
```

کلیه مشخصات Box را بارث برده و به آنها عنصر weight را اضافه می کند . برای Boxweight ضرورتی ندارد که کلیه جوانب موجود در Box را مجددا "ایجاد نماید . بلکه می تواند بسادگی Box را طوری گسترش دهد تا اهداف خاص خودش را تامین نماید .

یک مزیت عمده وراثت این است که کافیست فقط یکبار یک کلاس بالا ایجاد کنید که خصلتهای مشترک یک مجموعه از اشیائ را تعریف

نماید ، آنگاه می توان از آن برای ایجاد هر تعداد از زیر کلاسهای مشخص تر استفاده نمود . هر زیـر کـلاس مـی توانـد دقیقـا" بـا طبقـه بنـدی خودش تطبیق یابد . بعنوان مثال ، کلاس بعدی ، از Box بارث برده و یک خصلت رنگ (color) نیز در آن اضافه شده است .

```
// Here/ Box is extended to include color.
class ColorBox extends Box {
int color; // color of box

ColorBox(double w/ double h/ double d/ double c ){
width = w;
height = h;
depth = d;
color = c;
}
}
```

بیاد آورید که هرگاه یک کلاس بالا ایجاد نمایید که وجوه عمومی یک شی ئ را تعریـف کنـد ، مـی تـوان از آن کـلاس بـالا بـرای تشکیل کلاسهای تخصصی تر ارث برد . هر زیر کلاس خیلی ساده فقط خصلتهای منحصر بفرد خودش را اضافه مـی کنـد . ایـن مفهـوم کلـی وراثـت است . یک متغیر کلاس بالا می تواند به یک شی ئ زیر کلاس ارجاع نماید یک متغیـر ارجـاع مربـوط بـه یـک کـلاس بـالا را مـی تـوان بـه ارجاعی ، به هر یک از زیر کلاسهای مشتق شده از آن کلاس بالا ، منتسب نمود . در بسیاری از شرایط ، ایـن جنبـه از وراثـت کـاملا" مفیـد و سودمند است . بعنوان مثال ، مورد زیر را در نظر بگیرید :

```
class RefDemo {
public static void main(String args[] ){
Boxweight weightbox = new BoxWeight(3/ 5/ 7/ 8.37);
Box plainbox = new Box();
double vol;

vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of weightbox is " +
weightbox.weight);
System.out.println();

// assign BoxWeight reference to Box reference
plainbox = weightbox;

vol = plainbox.volume(); // OK/ volume ()defined in Box
System.out.println("Volume of plainbox is " + vol);
```

```
/* The following statement is invalid because plainbox
dose not define a weight member .*/

// System.out.println("Weight of plainbox is " + plainbox.weight
}
}
```

در اینجا weightbox یک ارجاع به اشیائ Boxweight است و plainbox یک ارجاع به اشیائ Box است . از آنجاییکه Boxweightیک زیر کلاس از Box است ، می توانplainbox را بعنوان یک ارجاع به شی ئ weightbox منتسب نمود . نکته مهم این است که نوع متغیر ارجاع و نه نوع شیئی که به آن ارجاع شده است که تعیین می کند کدام اعضائ قابل دسترسی هستند . یعنی هنگامیکه یک ارجاع مربوط به یک شی ئ زیر کلاس ، به یک متغیر ارجاع کلاس بالا منتسب می شود ، شما فقط به آن بخشهایی از شی ئ دسترسی دارید که توسط کلاس بالا تعریف شده باشند . بهمین دلیل است که plainbox نمی تواند به weight دسترسی داشته باشد حتی وقتی که به یک شی ئ Boxweight ارجاع می کند . اگر به آن فکر کنید ، آن را احساس می کنید زیرا یک کلاس بالا آگاهی و احاطه ای نسبت به موارد اضافه شده به زیرکلاس مربوطه اش نخواهد داشت . بهمین دلیل است که آخرین خط از کد موجود در قطعه قبلی از توضیح رج شده است . برای یک ارجاع Box امکان ندارد تا به فیلد weight دسترسی داشته

ایجاد یک سلسله مراتب چند سطحی (Multilevel)

می توانید سلسله مراتبی بسازید که شامل چندین لایه وراثت بدلخواه شما باشند. کاملا" موجه است که از یک زیر کلاس بعنوان کلاس بالای یک کلاس دیگر استفاده کنیم . بعنوان مثال اگر سه کلاس A ، B،وC و داشته باشیم آنگاه C می تواند یک زیر کلاس از B وB و یک زیر کلاس از A باشد . وقتی چنین شرایطی اتفاق می افتد ، هر زیر کلاس کلیه خصلتهای موجود در کلیه کلاس بالاهای خود را بارث می برد . در این شرایط ، Cکلیه جنبه هایB وA و را بارث می برد . در برنامه بعدی ، زیر کلاس Boxweight بعنوان یک کلاس بالا استفاده شده تا زیر کلاس تحت عنوان shipment را ایجاد نماید shipment . کلیه خصلتهای Boxweight وBox را به ارث برده و یک فیلد بنام cost به آن اضافه شده که هزینه کشتیرانی یک محموله را نگهداری می کند .

```
// Extend BoxWeight to include shipping costs.

// Start with Box.
class Box {
private double width;
private double height;
private double depth;

// construct clone of an object
Box(Box ob ){ // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}

// constructor used when all dimensions specified
Box(double w, double h, double d ){
width = w;
height = h;
depth = d;
+ }
+
+ // constructor used when no dimensions specified
+ Box (){
+ width =- 1; // use- 1 to indicate
+ height =- 1; // an uninitialized
```

```
depth =- 1; // box
}

// constructor used when cube is created
Box(double len ){
width = height = depth = len;
}

// compute and return volume
double volume (){
return width * height * depth;
}
}

// Add weight.
class BoxWeight extends Box {
double weight; // weight of box

// construct clone of an object
BoxWeight(BoxWeight ob ){ // pass object to constructor
super(ob);
weight = ob.weight;
}

// constructor used when all parameters are specified
BoxWeight(double w/ double h/ double d/ double m ){
super(w, h, d); // call superclass constructor
weight = m;
}

// default constructor
BoxWeight (){
super();
weight =- 1;
}

// constructor used when cube is created
BoxWeight(double len/ double m ){
```

```
super(len);
weight = m;
}
}

// Add shipping costs
class Shipment extends BoxWeight {
double cost;

// construct clone of an object
Shipment(Shipment ob ){ // pass object to constructor
super(ob);
cost = ob.cost;
}

// constructor used when all parameters are specified
BoxWeight(double w, double h, double d,
double m, double c ){
super(w, h, d); // call superclass constructor
cost = c;
}
// default constructor
Shipment (){
super();
cost =- 1;
}

// constructor used when cube is created
BoxWeight(double len, double m, double c ){
super(len, m);
cost = c;
}
}

class DemoShipment {
public static void main(String args[] ){
Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.41);
Shipment shipment2 = new Shipment(2, 3, 4, 0.76, 1.28);
```

```
double vol;

vol = shipment1.volume();
System.out.println("Volume of shipment1 is " + vol);
System.out.println("Weight of shipment1 is " + shipment1.weight);
System.out.println("Shipping cost :$" + shipment1.cost);
System.out.println();

vol = shipment2.volume();
System.out.println("Volume of shipment2 is " + vol);
System.out.println("Weight of shipment2 is " + shipment2.weight);
System.out.println("Shipping cost :$" + shipment2.cost);
}
}
```

خروجی این برنامه بصورت زیر می باشد :

```
Volume of shipment1 is 3000
Weight of shipment1 is 10
Shipping cost :$3.41

Volume of shipment2 is 24
Weight of shipment2 is 0.76
Shipping cost :$1.28
```

بدلیل وراثت ، shipment می تواند از کلاسهای تعریف شده قبلی Box و Boxweight و استفاده نماید و فقط اطلاعات اضافی که برای کاربرد خاص خودش نیاز دارد ، اضافه نماید . این بخشی از ارزش وراثت است . وراثت امکان استفاده مجدد از کدهای قبلی را بخوبی بوجود آورده است . این مثال یک نکته مهم دیگر را نشان می دهد super() : همواره به سازنده موجود در نزدیکترین کلاس بالا ارجاع می کند super() . در shipment ر سازنده Boxweight را فراخوانی میکند super(). در Boxweight ر سازنده موجود در Box را فراخوانی میکند. در یک سلسله مراتب کلاس ، اگر یک سازنده کلاس بالا نیازمند پارامترها باشد، آنگاه کلیه زیر کلاسها باید آن پارامترها را بالای خط (up the line) بگذرانند. این امر چه یک زیر کلاس پارامترهای خودش را نیاز داشته باشد چه نیاز نداشته باشد ، صحت خواهد داشت .

نکته : در مثال قبلی ، کل سلسله مراتب کلاس ، شامل Box ، Boxweight، و shipment و همگی در یک فایل نشان داده می شوند. این حالت فقط برای راحتی شما است . اما در جاوا ، هر یک از سه کلاس باید در فایلهای خاص خودشان قرار گرفته و جداگانه کامپایل شوند . در حقیقت ، استفاده از فایلهای جداگانه یک می و نه یک استثنائ در ایجاد سلسله مراتب کلاسهاست

وقتی که سازندگان فراخوانی می شوند constructors

وقتی یک سلسله مراتب کلاس ایجاد می شود ، سازندگان کلاسها که سلسله مراتب را تشکیل می دهند به چه ترتیبی فراخوانی می شوند ?

بعنوان مثال ، با یک زیر کلاس تحت نام B و یک کلاس بالا تحت نام A ، آیا سازنده A قبل از سازنده B فراخوانی میشود، یا بالعکس ؟ پاسخ این است که در یک سلسله مراتب کلاس ، سازندگان بترتیب مشتق شدنشان از کلاس بالا به زیر کلاس فراخوانی می شوند . بعلاوه چون ()super باید اولین دستوری باشد که در یک سازنده زیر کلاس اجرا می شود ، این ترتیب همانطور حفظ می شود ، خواه ()super استفاده شود یا نشود . اگر ()super استفاده نشود آنگاه سازنده پیش فرض یا سازنده بدون پارامتر هر یک از زیر کلاسها اجرا خواهند شد .

برنامه بعدی نشان می دهد که چه زمانی سازندگان اجرا می شوند :

```
// Demonstrate when constructors are called.

// Create a super class.
class A {
A (){
System.out.println("Inside A's constructor.")
}
}

// Create a subclass by extending class A.
class B extends A {
B (){
System.out.println("Inside B's constructor.")
}
}

// Create another subclass by extending B.
class C extends B {
C (){
System.out.println("Inside C's constructor.")
}
}

class CallingCons {
public static void main(String args[] ){
C c = new C();
}
}
```

خروجی این برنامه بشرح زیر می باشد :

```
Inside A's constructor
```

```
Inside B's constructor
Inside C's constructor
```

همانطوریکه مشاهده می کنید، سازندگان بترتیب مشتق شدنشان فراخوانی می شوند .اگر درباره آن تفکر کنید ، می فهمید که توابع سازنده بترتیب مشتق شدنشان اجرا می شوند . چون یک کلاس بالا نسبت به زیر کلاسهای خود آگاهی ندارد ، هر گونه مقدار دهی اولیه که برای اجرا شدن نیاز داشته باشد، جدا از و احتمالا" پیش نیاز هر گونه مقدار دهی اولیه انجام شده توسط زیر کلاس بوده است .

استفاده از Super

در مثالهای قبلی کلاسهای مشتق شده از Box به کارایی و قدرتمندی که امکان داشت ، پیاده سازی نشدند . بعنوان مثال ، سازنده Boxweightبطور صریحی فیلدهای width ، heightو depth را مقدار دهی اولیه می کند و در Box() . این امر نه تنها کدهای پیدا شده در کلاس بالای آنها را دو برابر می کند که غیر کاراست ، بلکه دلالت دارد بر اینکه یک زیر کلاس باید دسترسی به این اعضائ داشته باشد . اما شرایطی وجود دارند که می خواهید یک کلاس بالا ایجاد کنید که جزئیات پیاده سازی خودش را خودش نگهداری کند . در این شرایط ، راهی برای یک زیر کلاس وجود ندارد تا مستقیما"به این متغیرهای مربوط به خودش دسترسی داشته و یا آنها را مقداردهی اولیه نماید . از آنجاییکه کپسول سازی یک خصلت اولیه oop است ، پس باعث تعجب نیست که جاوا راه حلی برای این مشکل فراهم کرده باشد . هرگاه لازم باشد تا یک زیر کلاس به کلاس بالای قبلی خودش ارجاع نماید ، اینکار را با استفاده از واژه کلیدی super انجام می دهیم . superدو شکل عمومی دارد . اولین آن سازنده کلاس بالا را فراخوانی می کند . دومین آن بمنظور دسترسی به یک عضو کلاس بالا که توسط یک عضو زیر کلاس مخفی مانده است ، استفاده می شود .

استفاده از super

یک زیرکلاس میتواند روش سازنده تعریف شده توسط کلاس بالای مربوطه را بااستفاده از این شکل super فراخوانی نماید :

```
super( parameter-list);
```

در اینجا parameter-listمشخص کننده هر پارامتری است که توسط سازنده در کلاس بالا مورد نیاز باشد .super() باید همواره اولین دستور اجرا شده داخل یک سازنده زیر کلاس باشد . بنگرید که چگونه از super() استفاده شده ، و همچنین این روایت توسعه یافته از کلاس Boxweight() را در نظر بگیرید :

```
// BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
double weight; // weight of box

// initialize width/ height/ and depth using super()
BoxWeight(double w/ double h/ double d/ double m ){
super(w, h, d); // call superclass constructor
weight = m;
}
}
```

در اینجا ()Boxweight فراخوانی ()super را با پارامترهای w ، h،و d و انجام می دهد. این کار سبب فراخوانده شدن سازنده ()Box شده با استفاده از این مقادیر width، height، وdepth و را مقدار دهی اولیه می کند . دیگر Boxweight خودش این مقادیر اولیه را مقدار دهی نمی کند . فقط کافی است تا مقدار منحصر بفرد خودweight : را مقدار دهی اولیه نماید . این عمل Box را آزاد می گذارد تا در صورت تمایل این مقادیر را private بسازد .

در مثال قبلی ، ()superبا سه آرگومان فراخوانی شده بود . اما چون سازندگان ممکن است انباشته شوند ، می توان ()super را با استفاده از هر شکل تعریف شده توسط کلاس بالا فراخوانی نمود . سازنده ای که اجرا می شود ، همانی است که با آرگومانها مطابقت داشته باشد . بعنوان مثال ، در اینجا یک پیاده سازی کامل ازBoxweight وجود دارد که سازندگان را برای طرق گوناگون و ممکن ساخته شدن یک boxفراهم می نماید. در هر حالت ()super با استفاده از آرگومانهای تقریبی فراخوانی میشود. دقت کنید کهwidth وheight و depthو داخل Box بصورت اختصاصی درآمده اند .

```
// A complete implementation of BoxWeight.
class Box {
private double width;
private double heght;
private double deoth;

// construct clone of an object
Box(Box ob ){ // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}

// constructor used when all dimensions specified
Box(double w/ double h/ double d ){
width = w;
height = h;
depth = d;
}

// constructor used when no dimensions specified
Box (){
width =- 1; // use- 1 to indicate
height =- 1; // an uninitialized
depth =- 1; // box
}
```

```
// constructor used when cube is created
Box(double len ){
width = height = depth = len;
}

// compute and return volume
double volume (){
return width * height * depth;
}
}

// BoxWeight now fully implements all construtors.
class BoxWeight extends Box {
double weight; // weight of box

// construct clone of an object
BoxWeight(BoxWeight ob ){ // pass object to consructor
super(ob);
weight = ob.weight;
}

// constructor used when all parameters are specified
Box(double w, double h, double d, double m ){
super(w, h, d); // call superclass constructor

weight = m;
}

// default constructor
BoxWeight (){
super();
weight =- 1;
}

// constructor used when cube is created
BoxWeight(double len, double m ){
super(len);
```

```
weight = m;
}
}

class DemoSuper {
public static void main(String args[] ){
BoxWeight mybox1 = new BoxWeight(10 ,20 ,15 ,34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
BoxWeight mybox3 = new BoxWeight(); // default
BoxWeight mycube = new BoxWeight(3, 2);
BoxWeight myclone = new BoxWeight(mybox1);
double vol;

vol = mybox1.vilume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();

vol = mybox2.vilume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
System.out.println();

vol = mybox3.vilume();
System.out.println("Volume of mybox3 is " + vol);
System.out.println("Weight of mybox3 is " + mybox3.weight);
System.out.println();

vol = myclone.vilume();
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
System.out.println();

vol = mycube.vilume();
System.out.println("Volume of mycube is " + vol);
System.out.println("Weight of mycube is " + mycube.weight);
System.out.println();
}
```

```
}
```

این برنامه خروجی زیر را تولید می کند :

```
Volume of mybox1 is 3000
Weight of mybox1 is 34.3

Volume of mybox2 is 24
Weight of mybox2 is 0.076

Volume of mybox3 is- 1
Weight of mybox3 is- 1

Volume of myclone is 3000
Weight of myclone is 34.3

Volume of mycube is 27
Weight of mycube is 2
```

توجه بیشتری نسبت به این سازنده در Boxweight() داشته باشید :

```
// construct clone of an object
BoxWeight(BoxWeight ob ){ // pass object to constructor
super(ob);
weight = ob.weight;
}
```

توجه کنید که super() با یک شی ئاز نوع Boxweight نه از نوع Box فراخوانی شده است و نیز سازنده Box (Box ob) را فراخوانی می کند . همانطوریکه قبلا" ذکر شد ، یک متغیر کلاس بالا را می توان برای ارجاع به هر شی ئ مشتق شده از آن کلاس مورد استفاده قرار داد . بنابراین ، ما قادر بودیم یک شی ئ Boxweight را به سازنده Box گذر دهیم . البته Box فقط نسبت به اعضائ خودش آگاهی دارد . اجازه دهید مفاهیم کلیدی مربوط به super() را مرور نماییم . وقتی یک زیرکلاس super() را فراخوانی می کند ، در اصل سازنده کلاس بالای بلافصلخودرافراخوانی می کند . بنابراین super() همواره به کلاس بالای بلافصل قرار گرفته در بالای کلاس فراخوانده شده ، ارجاع میکند . این امر حتی در یک سلسله مراتب چند سطحی هم صادق است . همچنین super باید همواره اولین دستوری باشد که داخل یک سازنده زیر کلاس اجرا می شود .

دومین کاربرد super

دومین شکل super تا حدودی شبیه this کار می کند، بجز اینکه super همواره به کلاس بالای زیر کلاسی که در آن استفاده می شود ، ارجاع می کند . شکل عمومی این کاربرد بصورت زیر است :

Super .member

در اینجا ، memberممکن است یک روش یا یک متغیر نمونه باشد . این دومین شکل super برای شرایطی کاربرد دارد که در آن اسامی اعضائ یک زیر کلاس ، اعضائ با همان اسامی را در کلاس بالا مخفی می سازند . این سلسله مراتب ساده کلاس را در نظر بگیرید :

```
// Using super to overcome name hiding.
class A {
int i;
}

// Create a subclass by extending class A.
class B extends A {
int i; // this i hides the in A

B(int a,int b ){
super.i = a; // i in A
i = b; // i in B
}

void show (){
System.out.println("i in superclass :" + super.i);
System.out.println("i in subclass :" + i);
}
}

class UseSuper {
public static void main(String args[] ){
B subOb = new B(1,2);

subOb.show();
}
}
```

این برنامه خروجی زیر را نمایش می دهد :

i in superclass :1

i in subclass :2

اگرچه متغیر نمونهi درB ر متغیرi درA ر را پنهان می سازد ، اما super امکان دسترسی به i تعریف شده در کلاس بالا بوجود می آورد .

همانطوریکه خواهید دید همچنین میتوان از super برای فراخوانی روشهایی که توسط یک زیر کلاس مخفی شده اند

روش finalize()

گاهي لازم است تا يک شي ئ هنگاميکه در حال خراب شدن است ، يک عمل خاصي را انجام دهد. بعنوان مثال ، ممکـن اسـت يـک شـي ئ دربرگيرنده منابع غير جاوا نظير يک دستگيره فايل (file handle) يا فونت کاراکتر ويندوز باشد، و مي خواهيد اطمينـان يابيـد کـه قبـل از اينکه آن شي ئ خراب شود ، منابع فوق آزاد شوند . براي اداره چنين شرايطي ، جاوا مکانيسـمي تحـت نـام (finalization تمـام کننـده) فراهم آورده است . با استفاده از اين مکانيسم ، مي توانيد عمليات مشخصي را تعريف نماييد که زمانيکه يک شـي ئ در شـرف مرمـت شـدن توسط جمع آوري زباله است ، اتفاق بيفتند . براي افزودن يک تمام کننـده (finalizer) بـه يـک کـلاس ، خيلـي راحـت کـافي اسـت تـا روش finalize() را تعريف نماييد. سيستم حين اجراي جاوا هرگاه در شرف چـرخش مجـدد يـک شـي ئ از کلاسـي باشـد ، ايـن روش را فراخواني مي کند . داخل روش finalize() شما عملياتي را مشخص مي کنيد که بايد درست قبل از خرابي آن شي ئ انجام گيرنـد . جمـع کننده زباله (garbage collector) بطور متناوب اجرا شده و بدنبال اشيايي ميگردد که ديگر مورد ارجاع هيچيک از شرايط اجرايي نبوده و يا غير مستقيم توسط ساير اشيائ ارجاع شده باشند . درست قبل از اينکه يک داراي رها شود، سيستم حين اجراي جاوا روش finalize() را روي شي ئ فراخواني مي کند . شکل عمومي روش finalize() بقرار زير مي باشد :

```
protected void finalize()
{
// finalization code here
}
```

دراينجا واژه کليدي protected توصيفگري است که از دسترسي بـه روش finalize() توسـط کـدهاي تعريـف شـده خـارج از کـلاس جلوگيري مي کند .مهم است بدانيد که روش finalize() درست مقدم بر جمع آوري زباله فراخواني مي شود . بعنوان مثال وقتي يـک شـي ئ از قلمرو خارج مي شود ، اين روش فراخواني نخواهد شد.اين بدان معني است که نميتوانيد تشخيص بدهيد که چه زماني finalize() اجرا خواهد شد و يا اصلا" اجرا نخواهد شد . بنابراين ، برنامه شما بايد ساير وسائل براي آزاد کردن منـابع سيسـتم مـورد اسـتفاده شـي ئ را تـدارک ببيند . برنامه نبايد براي عمليات برنامه اي عادي ، متکي به روش finalize() باشد . نکته : اگر با C++ آشنايي داريد، پس مي دانيد کـه C++امکان مي دهد تا يک خراب کننده (destructor) براي يک کلاس تعريف نماييد ، که هرگاه يک شي ئ خارج از قلمـرو قـرار گيرد ، فراخواني خواهد شد . جاوا چنين کاري نکرده و از اين ايده پشتيباني نمي کند . روش finalize() فقط بـه يـک تـابع خـراب کننـده نزديک مي شود. به مرور که تجربه بيشتري با جاوا کسب مي کنيد ، مي بينيد که نياز به توابع خراب کننده بسيار کم است زيرا زير سيستم جمع آوري حسن انجام مي دهد.

توزیع (dispatch) پویای روش

اگر در لغو روشها چیزی فراتر از یک قرارداد فضای نام وجود نداشت ، آنگاه این عمل در بهترین حالت ، ارضـائ نـوعی حـس کنجکـاوی و فاقد ارزش عملی بود . اما این چنین نیست . لغو روش تشکیل دهنده اساس یکی از مفاهیم پرقدرت در جاوا یعنی" توزیع پویای روش " است . این یک مکانیسم است که توسط آن یک فراخوانی به تابع لغو شده در حین اجرا (در عوض زمـان کامپایـل) از سـر گرفتـه مـی شـود. توزیـع پویای روش مهم است چون طریقی است که جاوا با آن چند شکلی را درست حین اجرا پیاده سازی می نماید . توضیح را با تکرار یک اصـل مهم شروع میکنیم : یک متغیر ارجاع کلاس بالا میتواند به یک شی ئ زیر کلاس ارجاع نماید . جاوا از این واقعیت استفاده کرده و فراخوانی به روشهای لغو شده را حین اجرا از سر می گیرد . وقتی یک روش لغو شده از طریق یک ارجاع کلاس بالا فراخوانی می شـود، جـاوا براسـاس نوع شی ئ ارجاع شده در زمانی که فراخوانی اتفاق می افتد ، تعیین می کند که کدام روایت از روش باید اجرا شـود . بنـابراین ، عمـل تعیـین روایت خاص از یک روش ، حین اجرا انجام می گیرد . وقتی به انواع مختلف اشیائارجاع شده باشد، روایتهای مختلفی از یک روش لغـو شـده فراخوانی خواهند شد . بعبارت دیگر ، این نوع شی ئ ارجاع شده است (نه نوع متغیر ارجاع) که تعیین می کنـد کـدام روایـت از روش لغـو شده باید اجرا شود . بنابراین اگر یک کلاس بالا دربرگیرنده یک روش لغو شده توسط یک زیر کلاس باشد ، آنگاه زمانی که انـواع مختلـف اشیائ از طریق یک متغیر ارجاع کلاس بالا مورد ارجاع قرار می گیرنـد روایتهـای مختلـف آن روش اجـرا خواهنـد شـد . در اینجـا مثـالی را مشاهده میکنید که توزیع پویای روش را به شما نشان میدهد :

```
// Dynamic Method Dispatch
class A {
void callme (){
System.out.println("Inside A's callme method");
}
}

class B extends A {
// override callme()
void callme (){
System.out.println("Inside B's callme method");
}
}

class C extends A {
// override callme()
void callme (){
System.out.println("Inside C's callme method");
}
```

```
}

class Dispatch {
public static void main(String args[] ){
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A

r = a; // r refers to an A object
r.callme(); // calls A's version of callme

r = b; // r refers to a B object
r.callme(); // calls B's version of callme

r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

خروجی این برنامه بقرار زیر می باشد :

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

این برنامه یک کلاس بالای تحت نام A و دو زیر کلاس آن تحت نامB وC و را ایجاد می کند. زیر کلاسهایB وC و سبب لغو callme()اعلان شده در A می گردند. درون روش main() اشیائی از نوعA وB وC و اعلان شده اند . همچنین یک ارجاع از نوع A بنامr اعلان شده است . سپس برنامه یک ارجاع به هر یک از انواع اشیائ به r را نسبت داده و از آن ارجاع برای فراخوانی callme() استفاده می کند . همانطوریکه حاصل این برنامه نشان می دهد ، روایتی از callme() که باید اجرا شود توسط نوع شیئی که در زمان فراخوانی مورد ارجاع قرار گرفته ، تعیین می شود . اگر این تعیین توسط نوع متغیر ارجاع یعنی r انجام میگرفت شما با سه فراخوانی به روش callme()

مربوط به A مواجه می شدید . نکته : کسانی که با C++ آشنا هستند تشخیص می دهند که روشهای لغو شده در جاوا مشابه توابع مجازی (virtual functions)در C++ هستند

.

چرا روشهای لغو شده ?

قبلا" هم گفتيم كه روشهاي لغو شده به جاوا اجازه پشتيباني از چند شكلي حين اجرا را مي دهند. چند شكلي به يك دليل براي برنامه نويس شي ئ گرا لازم است : اين حالت به يك كلاس عمومي اجازه مي دهد تا روشهايي را مشخص نمايد كه براي كليه مشتقات آن كلاس مشترك باشند ، و به زير كلاس ها اجازه مي دهد تا پيادهسازيهاي مشخص برخي يا كليه روشها را تعريف نمايند. روشهاي لغو شده راه ديگري براي جاوا است تا " يك ابط و چندين روش " را بعنوان يكي از وجوه چند شكلي پياده سازي نمايد .

بخشي از كليد كاربرد موفقيت آميز چند شكلي ، درك اين نكته است كه كلاس بالاها و زير كلاسها يك سلسله مراتب تشكيل ميدهند كه از مشخصات كوچكتر به بزرگتر حركت مي كنند. اگر كلاس بالا بدرستي استفاده شود، كليه اجزائي كه يك زير كلاس مي تواند بطور مستقيم استفاده نمايد ، تعريف مي كند. اين امر به زير كلاس قابليت انعطاف تعريف روشهاي خودش را مي دهد ، كه همچنان يك رابط منسجم را بوجود مي آورد .

بنابراين ، بوسيله تركيب وراثت با روشهاي لغو شده ، يك كلاس بالا مي تواند شكل عمومي روشهايي را كه توسط كليه زير كلاسهاي مربوطه استفاده خواهند شد را تعريف نمايد .

چند شكلي پويا و حين اجرا يكي از قدرتمندترين مكانيسمهايي است كه طراحي شي ئ گرايي را مجهز به استفاده مجدد و تنومندي كدها نموده است . اين ابزار افزايش دهنده قدرت كتابخانه هاي كدهاي موجود براي فراخواني روشهاي روي نمونه هاي كلاسهاي جديد بدون نياز به كامپايل مجدد مي باشد در حاليكه يك رابط مجرد و زيبا را نيز حفظ مي كنيم .

بكار بردن لغو روش

اجازه دهيد تا به يك مثال عملي تر كه از لغو روش استفاده مي كند ، نگاه كنيم . برنامه بعدي يك كلاس بالا تحت نام Figure را ايجاد مي كند كه ابعاد اشيائ مختلف دو بعدي را ذخيره مي كند . اين برنامه همچنين يك روش با نام ()area را تعريف مي كند كه مساحت يك شي ئ را محاسبه مي كند. برنامه ، دو زير كلاس از Figure مشتق مي كند . اولين آن Rectangle و دومين آن Triangle است . هر يك از اين زير كلاسها ()area را طوري لغو ميكنند كه بترتيب مساحت يك مستطيل و مثلث را برگردان كنند .

```
// Using run-time polymorphism.
class Figure {
double dim1;
double dim2;
```

```
Figure(double a/ double b ){
dim1 = a;
dim2 = b;
}

double area (){
System.out.println("Area for Figure is undefined.");
return 0;
}
}

class Rectangle extends Figure {
Rectangle(double a,double b ){
super(a,b);
}

// override area for rectangle
double area (){
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}

class Triangle extends Figure {
Triangle(double a,double b ){
super(a,b);
}

// override area for right triangle
double area (){
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}

class FindAreas {
public static void main(String args[] ){
Figure f = new Figure(10/ 10);
```

```
Rectangle r = new Rectangle(9/ 5);
Triangle t = new Triangle(10/ 8);

Figure figref;

figref = r;
System.out.println("Area is " + figref.area)();

figref = t;
System.out.println("Area is " + figref.area)();

figref = f;
System.out.println("Area is " + figref.area)();
}
}
```

حاصل این برنامه بقرار زیر است :

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

از طریق مکانیسم دوگانه وراثت و چند شکلی حین اجرا ، امکان تعریف یک رابط منسجم که برای چندین نوع اشیائ مختلف ، اما بهم مرتبط ، استفاده می شود ، وجود دارد . در این حالت ، اگر از Figure یک شی ئ مشتق شود ، پس با فراخوانی area() می توان مساحت آن شی ئ را بدست آورد . رابط مربوط به این عملیات صرفنظر از نوع می باشد.

واژه کلیدی This

گاهي لازم است يك روش به شي ئ كه آن را فراخواني نموده ، ارجاع نمايد . براي اين منظور، جاوا واژه كليدي this را تعريف كرده است . this را مي توان داخل هر روشي براي ارجاع به شي ئ جاري (current) استفاده نمود .يعني this همواره ارجاعي است به شي ئ كه روش روي آن فراخواني شده است . مي توانيد از this هر جايي كه ارجاع به يك شي ئ از نوع كلاس جاري مجاز باشد ، استفاده نماييد . براي اينكه بفهميد this به چه چيزي ارجاع مي كند ، روايت بعدي Box() را در نظر بگيريد :

```
// A redundant use of this.
Box(double w/ double h/ double d ){
this.width = w;
this.height = h;
this.depth = d;
}
```

اين روايت Box() دقيقا" مثل روايت اوليه كار مي كند . استفاده از this به ندرت انجام گرفته اما كاملا"صحت دارد. داخل Box() همواره this به شي ئ فراخوانده شده ارجاع مي كند . اگرچه در اين مثالها بندرت پيش آمده مي كند . اگرچه در اين مثالها بندرت پيش آمده ، اما this در ساير متون بسيار مفيد است .

مخفي نمودن متغير نمونه

حتما" مي دانيد كه اعلان دو متغير محلي با يك نام داخل يك قلمرو يا قلمروهاي بسته شده ، غير مجاز است . بطرز خيلي جالبي ، مي توانيد متغيرهاي محلي شامل پارامترهاي رسمي براي روشها ، داشته باشيد كه با اسامي متغيرهاي نمونه كلاسها مشترك باشند . اما ، هنگاميكه يك متغير محلي همان اسم متغير نمونه را داشته باشد، متغير محلي ، متغير نمونه را مخفي مي سازد .بهمين دليل بود كه از width height و depthو بعنوان اسامي پارامترهاي سازنده Box() داخل كلاس Box استفاده نكرديم . اگر از آنها بهمين روش استفاده مي كرديم ، آنگاه widthبه پارامتر رسمي ارجاع مي كرد و متغير نمونه width را مخفي مي ساخت . اگرچه آسان تر است كه از اسامي متفاوت استفاده كنيم ، اما راه حل ديگري براي چنين شرايطي وجود دارد . چون this امكان ارجاع مستقيم به شي ئ را به شما مي دهد ، مي توانيد با استفاده از آن هر نوع تصادف و يكي شدن اسامي بين

متغيرهاي نمونه و متغيرهاي

محلی را رفع نمایید . بعنوان مثال ، در اینجا روایت دیگری از Box() وجود دارد که از width ، height،وdepth و بعنوان اسامی

پارامترها استفاده نموده و آنگاه از this برای دستیابی به متغیرهای نمونه با همین اسامی استفاده کرده است :

```
// Use this to resolve name-space collisions.
Box(double width/ doublee height/ double depth ){
this.width = width;
this.height = height;
this.depth = depth;
}
```

احتیاط . استفاده از this درچنین متنی ممکن است گاهی گیج کننده بشود و برخی از برنامه نویسان مراقب هستند تا از اسامی متغیرهای محلی و پارامترهای رسمی که متغیرهای نمونه را مخفی می سازند ، استفاده نکنند. البته ، سایر برنامه نویسان طور دیگری فکر میکنند . یعنی معتقدند استفاده از اسامی مشترك برای وضوح ، ایده خوبی است و از this برای غلبه بر مخفی سازی متغیر نمونه بهره میگیرند. انتخاب یکی از دو روش با سلیقه شما ارتباط دارد . اگرچه this در مثالهایی که تاکنون نشان داده ایم ارزش زیادی نداشته ، اما در واهد بود .

استفاده از کلاسهای مجرد (abstract)

شرایطی وجود دارد که میخواهید یک کلاس بالا تعریف نمایید که ساختار یک انتزاع معین را بدون یک پیاده سازی کامل از هر روشی ، اعلان نماید. یعنی گاهی می خواهید یک کلاس بالا ایجاد کنید که فقط یک شکل عمومی شده را تعریف کند که توسط کلیه زیر کلاسهایش باشتراک گذاشته خواهد شد و پر کردن جزئیات این شکل عمومی بعهده هر یک از زیر کلاس ها واگذار می شود . یک چنین کلاسی طبیعت روشهایی که زیر کلاسها باید پیاده سازی نمایند را تعریف می کند . یک شیوه برای وقوع این شرایط زمانی است که یک کلاس بالا توانایی ایجاد یک پیاده سازی با معنی برای یک روش را نداشته باشد . تعریف area() خیلی ساده یک نگهدارنده مکان (place holder)است . این روش مساحت انواع شی ئ را محاسبه نکرده و نمایش نمی دهد . هنگام ایجاد کتابخانه های خاص کلاس خود ، خواهید دید که غیر معمول نیست اگر یک روش هیچ تعریف بامعنی در متن (context) کلاس بالای خود نداشته باشد . این شرایط را بدو طریق می توانید اداره نمایید . یک طریق این است که یک پیام هشدار (warning) گزارش نمایید . اگرچه این روش در برخی شرایط خاص مثل اشکال زدایی(debugging) فید است ، اما روش دائمی نیست . ممکن است روشهایی داشته باشید که باید توسط زیر کلاس لغو شوند تا اینکه آن زیرکلاس معنادار بشود .کلاس Triangle را در نظر بگیرید . اگر area() تعریف نشود، این کلاس هیچ معنایی ندارد . در این حالت ، شما بدنبال راهی هستید تا مطمئن شوید که یک زیر کلاس در حقیقت کلیه روشهای ضروری را لغو می کند . راه حل جاوا برای این مشکل روش مجرد abstract) (method است . می توانید توسط زیر کلاسها و با مشخص نمودن اصلاح کننده نوعabstract ، روشهای معینی را لغو نمایید . به این روشها گاهی subclasser responsibilty اطلاق میشود زیرا آنها هیچ پیاده سازی مشخص شده ای در کلاس بالا ندارند . بنابراین یک زیرکلاس باید آنها را لغو نماید چون نمی تواند بسادگی روایت تعریف شده در کلاس بالا را استفاده نماید . برای اعلان یک روش مجرد ، از شکل عمومی زیر استفاده نمایید .

```
abstract type name( parameter-list);
```

همانطوریکه مشاهده می کنید در اینجا بدنه روش معرفی نشده است . هر کلاسی که دربرگیرنده یک یا چند روش مجرد باشد ، باید بعنوان مجرد اعلان گردد . برای اعلان یک کلاس بعنوان مجرد ، بسادگی از واژه کلیدی abstract در جلوی واژه کلیدی class در ابتدای اعلان کلاس استفاده می نمایید . برای یک کلاس مجرد هیچ شیئی نمی توان ایجاد نمود . یعنی یک کلاس مجرد نباید بطور مستقیم با عملگر newنمونه سازی شود . چنان اشیائی بدون استفاده هستند ، زیرا یک کلاس مجرد بطور کامل تعریف نشده است . همچنین نمی توانید سازندگان مجرد یا روشهای ایستای مجرد اعلان نمایید . هر زیر کلاس از یک کلاس مجرد باید یا کلیه روشهای مجرد موجود در کلاس بالا را پیاده سازی نماید ، و یا خودش بعنوان یک abstract اعلان شود . در اینجا مثال ساده ای از یک کلاس با یک روش مجرد مشاهده می کنید که بعد از آن یک کلاس قرار گرفته که آن روش را پیاده سازی می کند :

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
```

```
// concrete methods are still allowed in abstract classes
void callmetoo (){
System.out.println("This is a concrete method.");
}
}

class B extends A {
void callme (){
System.out.println("B's implementetion of callme.");
}
}

class AbstractDemo {
public static void main(String args[] ){
B b = new B();

b.callme();

b.callmetoo();
}
}
```

توجه کنید که هیچ شیئی از کلاس A در برنامه اعلان نشده است . همانطوریکه ذکر شد ، امکان نمونه سازی یک کلاس مجرد وجود نـدارد . یک نکته دیگر : کلاس A یک روش واقعی با نام callmetoo() را پیاده سازی می کند . این امر کاملا" مقبول است . کلاسـهای مجـرد می توانند مادامیکه تناسب را حفظ نمایند ، دربرگیرنده پیاده سازیها باشند .

اگرچه نمی توان از کلاسهای مجرد برای نمونه سازی اشیائ استفاده نمود، اما از آنها برای ایجاد ارجاعات شی ئ می توان اسـتفاده نمـود زیـرا روش جاوا برای چند شکلی حین اجرا از طریق استفاده از ارجاعات کلاس بالا پیاده سازی خواهد شد . بنابراین ، باید امکان ایجاد یک ارجاع به یک کلاس مجرد وجود داشته باشد بطوریکه با استفادهاز آن ارجاع به یک شی ئ زیر کلاس اشاره نمود . شما اسـتفاده از ایـن جنبـه را در مثال بعدی خواهید دید . با استفاده از یک کلاس مجرد، می توانید کلاس Figure را توسعه دهید. چون مفهوم با معنایی برای مساحت یک شکل دو بعدی تعریف نشده وجود ندارد ، روایت بعدی این برنامه area() را بعنوان یک مجرد داخل Figure اعلان مـی کنـد . ایـن البتـه بدان معنی است که کلیه کلاسهای مشتق شده از Figure باید area() را لغو نمایند .

```
// Using abstract methods and classes.
abstract class Figure {
double dim1;
double dim2;

Figure(double a/ double b ){
dim1 = a;
dim2 = b;
}

// area is now an abstract method
abstruct double area();
}
class Rectangle extends Figure {
Rectangle(duoble a,double b ){
super(a,b);
}

// override area for rectangle
double area (){
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}

class Triangle extends Figure {
Triangle(double a,double b ){
super(a,B);
}

// override area for right triangle
double area (){
System.out.println("Inside Area for Teriangle.");
return dim1 * dim2 / 2;
}
}

class AbstractAreas {
```

```
public static void main(String args[] ){
// Figure f = new Figure(10/ 10); // illegal now
Rectangle r = new Rectanlge(9/ 5);
Triangle t = new Triangle(10/ 8);

Figure figref; // this is OK/ no object is created

figref = r;
System.out.println("Area is " + figref.area)();

figref = t;
System.out.println("Area is " + figref.area)();
}
}
```

همانطوریکه توضیح درون main() نشان می دهد ، دیگر امکان اعلان اشیائ از نوعFigure وجود ندارد چون اکنون بصورت مجرد است کلیه زیر کلاسهای Figure بایدarea() را لغو نمایند . برای اثبات این امر، سعی کنید یک زیر کلاس ایجاد نمایید که area() را لغو نمی کند . حتما" یک خطای complle-time دریافت می کنید . اگرچه امکان ایجاد یک شی ئاز نوع Figure وجود ندارد، اما می توانید یک متغیر ارجاع از نوع Figure ایجاد نمایید . متغیر ??? بعنوان ارجاعی به Figure اعلان شده و بدان معنی است که با استفاده ازآن می توان به یک شی ئ از هر کلاس مشتق شده از Figure ، ارجاع نمود . همانطوریکه توضیح دادیم ، از طریق متغیرهای ارجاع

درک مفهوم static

شرایطی وجود دارد که مایلید یک عضو کلاس را طوری تعریف کنید که مستقل از هر شی ئ آن کلاس مورد استفاده قرار گیرد. بطور معمول یک عضو کلاس باید فقط همراه یک شی ئ از همان کلاس مورد دسترسی قرار گیرد . اما ، این امکان وجود دارد که عضوی را ایجاد کنیم که توسط خودش استفاده شود ، بدون اینکه به یک نمونه مشخص ارجاع نماید . برای ایجاد چنین عضوی ، قبل از اعلان آن واژه کلیدی staticرا قرار دهید . وقتی یک عضو بعنوان static اعلان می شود ، می توان قبل از ایجاد هر شی ئ از آن کلاس ، و بدون ارجاعی به هیچیک از اشیائ ، آن را مورد استفاده قرار داد . می توانید هم روشها و هم متغیرها را بعنوان static اعلان نمایید. رایجترین مثال برای یک عضو static ، همان main() است main() . بعنوان static اعلان می شود چون باید قبل از وجود هر نوع شیئی فراخوانی شود. متغیرهای نمونه اعلان شده بعنوان static ضرورتا" متغیرهای سراسری هستند . هنگامیکه اشیائ کلاس آن اعلان می شوند ، هیچ کپی از متغیر static ساخته نمی شود .در عوض ، کلیه نمونه های آن کلاس همان متغیر static را باشتراک می گذارند . روشهای اعلان شده بعنوان static دارای چندین محدودیت هستند : ؤ آنها فقط سایر روشهای static را فراخوانی می کنند . ؤ آنها فقط به داده های static دسترسی دارند . ؤ آنها بهیچ وجه امکان ارجاع به this و super و را ندارند . اگر لازم است محاسباتی انجام دهید تا متغیرهای static را مقدار دهی اولیه نمایید ، می توانید یک بلوک static اعلان نمایید که فقط یکبار آنهم زمانی که کلاس برای اولین مرتبه بارگذاری می شود ، اجرا گردد . مثال بعدی کلاسی را نشان میدهد که یک روش static برخی متغیرهای static و یک بلوک مقداردهی اولیه static دارد :

```
// Demonstrate static variables/ methods/ and blocks.
class UseStatic {
static int a = 3;
static int b;

static void meth(int x ){
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}

static {
System.out.println("Static block initialized.");
b = a * 4;
}
```

```
public static void main(String args[] ){
meth(42);
}
}
```

بمحض اینکه کلاس usestatic بارگذاری شود ، کلیه دستورات static اجرا میشوند. ابتدا a برابر 3 قرار گرفته ، سپس بلوک static اجرا شده (یک پیام را چاپ می کند) و در نهایت مقدار a*4 یا 12در b ر بعنوان مقدار اولیه نهاده می شود . سپس main() فراخوانی می شود که math() را فراخوانی نموده و مقدار 42را به x می گذرانـد . سـه دسـتور println() بـه دو متغیـر static یعنـی a وb و و همچنین به متغیر محلی x ارجاع می کنند . یادآوری : ارجاع به هر یـک از متغیرهـای نمونـه داخـل یـک روش static غیرمجـاز اسـت .

خروجی برنامه فوق بشرح زیر می باشد :

```
Static block initialized.
x = 42
a = 3
b = 12
```

خارج از کلاسی که تعریف شده اند ، روشها و متغیرهای static را می توان مستقل از هر نوع شی ئ مورد استفاده قرار داد . برای انجام اینکار ، فقط کافی است نام کلاس را با یک عملگر نقطه ای بعد از آن مشخص نمایید . بعنوان مثال ، اگر بخواهید یک روش static را از خارج کلاس مربوطه فراخوانی کنید ، با استفاده از شکل عمومی زیر اینکار را انجام می دهید :

```
classname.method()
```

در اینجا classname نام کلاسی است که روش static در آن اعلان شده است . همان طوری که می توانید ببینید ، این شکل بندی مشابه همان است که برای فراخوانی روش های غیر staticاز طریق متغیرهای ارجاع شی ئ انجام میگرفت . یک متغیر static را نیـز مـی تـوان بـا همان روش با استفاده از عملگر نقطه ای روی نام کلاس مورد دسترسی قرار داد. این روشی است که جاوا بوسیله آن یک روایت کنتـرل شـده از توابع سراسری و متغیرهای سراسری را پیاده سازی می کند . در اینجا یک مثال وجود دارد . داخل main() روش classname() و متغیر b کهstatic هستند در خارج از کلاسهای خود مورد دسترسی قرار می گیرند .

```
+ class StaticDemo {
+ static int a = 42;
+ static int b = 99;
+ static void callme (){
+ System.out.println("a = " + a);
+ }
+ }
```

```
+
+ class StaticByName {
+ public static void main(String args[] ){
+ StaticDemo.callme();
+ System.out.println("b + " + StaticDemo.b);
+ }
+ }
```

خروجی این برنامه بقرار زیر خواهد بود:

استفاده از اشیائ بعنوان پارامترها

تاکنون فقط از انواع ساده بعنوان پارامترهای روشها استفاده کرده ایم . اما هم صحیح و هم معمول است که اشیائ را نیز به روشها گذر دهیم .

بعنوان مثال برنامه ساده بعدی را در نظر بگیرید :

```
// Objects may be passed to methods.
class Test {
int a/ b;
Test(int , int j ){
 a= i;
b = j;
}

// return true if o is equal to the invoking object
boolean equals(Test o ){
if(o.a == a && o.b == b )return true;
else return false;
}
}

class PassOb {
public static void main(String args[] ){
Test ob1 = new Test(100, 22);

Test ob2 = new Test(100, 22);
Test ob3 = new Test-(1,- 1);

System.out.println("ob1 == ob2 :" + ob1.equals(ob2));
System.out.println("ob1 == ob3 :" + ob1.equals(ob3));
}
}
```

این برنامه ، خروجی بعدی را تولید می کند :

```
ob1 == ob2 :true
ob1 == ob3 :false
```

همانطوریکه مشاهده میکنید، روش ()equals داخل Test دو شی ئ را از نظر کیفیت مقایسه نموده و نتیجه را برمی گرداند . یعنی این روش ، شی ئ فراخواننده را با شیئی که گذر کرده مقایسه می کند . اگر محتوی آندو یکسان باشد ، آنگاه روش true را برمی گرداند . در

غیر اینصورت false را برمی گرداند . توجه داشته باشید که پارامتر o در ()equals مشخص کننده Test بعنوان نوع آن می باشد. اگرچه Test یک نوع کلاس ایجاد شده توسط برنامه است ،اما بعنوان انواع توکار جاوا و بهمان روش مورد استفاده قرار گرفته است . یکی از رایجترین کاربردهای پارامترهای شیئی مربوط به سازندگان است . غالبا "ممکن است بخواهید یک شی ئ جدید را بسازید طوری که این شی ئ در ابتدا نظیر یک شی ئ موجود باشد. برای انجام اینکار باید یک تابع سازنده تعریف نمایید که یک شی ئ از کلاس خود را بعنوان یک پارامتر انتخاب می کند . بعنوان مثال ، روایت بعدی از()Box به یک شی ئ امکان داده تا آغازگر دیگری باشد :

```
// Here/ Box allows one object to initialize another.
class Box {
double width;
double height;
double depth;

// construct clone of an object
Box(Box ob ){ // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}

// constructor used when all dimensions specified
Box(double w, double h, double d ){
width = w;
height = h;
depth = d;
}

// constructor used when no dimensions specified
Box (){
width =- 1; // use- 1 to indicate
height =- 1; // an uninitialized
depth =- 1; // box
}

// constructor used when cube is created
Box(double len ){
width = height = depth
}
```

```
// compute and return volume
double volume (){
return width * height * depth;
}
}

class OverloadCons2 {
public static void main(String args[] ){
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);

Box myclone = new Box(mybox1);

double vol;

// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);

// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);

// get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);

// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

بعدا" خواهید دید که وقتی شروع به ایجاد کلاسهای خود می نمایید، معمولا" برای اینکه اجازه دهیم تا اشیائ بصورتی موثر و آسان ساخته شوند ، لازم است اشکال های سازنده را فراهم آوریم .

استفاده از instanceof

گاهی خوب است که طی زمان اجرا ، نوع شی ئ را بدانیم . بعنوان مثال ، ممکن است یک نخ از اجرا داشته باشید که انواع گوناگونی از اشیائ تولید نموده و همچنین نخی که این اشیائ را پردازش می کند . در این شرایط ، برای نخ پردازنده مفید است که نوع هر یک از اشیایی را که به آنها می رسد ، بداند . شرایط دیگری که در آن دانستن نوع شی ئ در زمان اجرا مهم است ، تبدیل casting می باشد . در جاوا یک تبدیل نامعتبر و غیر مجاز سبب بروز خطای حین اجرا می شود . بسیاری از تبدیلات غیر مجاز را در زمان کامپایل می توان گرفت . اما تبدیل cast که شامل سلسله مراتب کلاس باشد می تواند تبدیلات غیرمجاز تولید کند که فقط در زمان اجرا قابل کشف هستند . بعنوان مثال ، یک کلاس بالا موسوم به A می تواند دو زیر کلاس B کنیم ، یا یک شی ئ C را به نوع A . اما مجاز نیستیم یک شی ئ B را به نوع C (یا بالعکس) تبدیل نماییم . از آنجاییکه یک شی ئ از نوع A می تواند به اشیائ B یاC ارجاع نماید ، در زمان اجرا چگونه می توان فهمید که به چه نوع شیئی ارجاع شده است ، قبل از اینکه تلاش برای تبدیل به نوع C را انجام دهیم ؟ آن شی ئ ممکن است شیئی از نوعA وB وC و باشد . اگر از نوع B باشد ، یک استثنائ زمان اجرا پرتاب خواهد شد . جاوا عملگر حین اجرای instanceof را برای پاسخگویی به همین سوال تدارک دیده است شکل عمومی عملگر instanceof بقرار زیر می باشد :

object instanceof type

در اینجا ، objectیک نمونه از کلاس است و type یک نوع کلاس است . اگر object از نوع مشخصی باشد و یا قابل تبدیل به یک نوع مشخص شده باشد ، آنگاه عملگرinstanceof مقدار true را نشان میدهد. در غیر اینصورت ، منجر به false میگردد. بدین ترتیب ، instanceofوسیله ای است که توسط آن برنامه اتان می تواند اطلاعات نوع درباره یک شی ئ در زمان اجرا را بدست آورد . برنامه بعدی نشان دهنده instanceof می باشد :

```
// Demonstrate instanceof operator.
class A {
int i/ j;
}

class B {
int i/ j;
}

class C extends A {
int k;
}
```

```java
class D extends A {
int k;
}
class InstanceOf {
public static void main(String args[] ){
A a = new A();
B b = new B();
C c = new C();
D d = new D();

if(a instancof A)
System.out.println("a is instance of A");
if(b instancof B)
System.out.println("b is instance of B");
if(c instancof C)
System.out.println("c is instance of C");
if(c instancof A)
System.out.println("c is instance of A");
if(a instancof C)
System.out.println("a is instance of C");

System.out.println();

// compare types of derived types

A ob;

ob = d; // A reference to d
System.out.println("ob new refers to d");
if(ob instancof D)
System.out.println("ob is instance of D");

System.out.println();

ob = c; // A reference to c
System.out.println("ob new refers to c");
if(ob instancof D)
System.out.println("ob is instance of D");
```

```
else
System.out.println("ob is instance of D");

if(ob instancof A)
System.out.println("ob is instance of A");

System.out.println();

// all object can be cast to Object
if(a instancof object)
System.out.println("a may be cast to Object");
if(b instancof object)
System.out.println("b may be cast to Object");
if(c instancof object)
System.out.println("c may be cast to Object");
if(d instancof object)
System.out.println("d may be cast to Object");
}
}
```

خروجی حاصل از این برنامه بصورت زیر می باشد :

```
a is instance of A
b is instance of B
c is instance of C
c can be cast to A

ob now refers to d
ob is instance of D

ob now refers to c
ob cannot be cast to D
ob can be cast to A

a may be cast to Object
b may be cast to Object
c may be cast to Object
d may be cast to Object
```

عملگر instanceof برای اکثر برنامه ها مورد نیاز نیست ، زیر معمولا" شما نوع شیئی را که با آن کار می کنید ، می دانید . اما ، وقتی مشغول نوشتن روالهای عمومی شده هستید که با یک شی ئ از یک سلسله مراتب کلاس پیچیده عمل می کند ، این عملگر بسیار مفید خواهد بود .

کلاس Singletone

گاهی به کلاس هایی بر می خوریم که لزوما باید یک و فقط یک متغیر از آنها تعریف شود مثلا یک عامل یا شیئ کـه بـه یـک منبع غیـر قابـل اشترک دسترسی دارد اما هیچ چیزی نمی تواند شی را از تعریف متغیر دیگری از آن باز دارد پس چه می شود کرد الگوی تک برگ پاسخی به این پرسش است الگوی تک برگ با گرفتن وظیفه ایجاد و قطع دسترسی به متغیر در خود شی طرح را محدود می کند چنین کاری تضمین می کند که تنها یک کتغیر ایجاد شود و دسترسی به آن منفرد باشد.

پیاده سازی الگوی تک برگ :

```
/*
  a class refrence to the singleton instance
*/
public  class Singleton {
        private static Singleton instance;
        protected Singleton(){}
        public static Singleton getInstance(){
            if (instance== null) {
                instance= new Singleton();
              }
            return instance;
        }


}
```

کلاس Singleton یک متغیر Static از نوع Singleton دارد که دسترسی به آن را فقط به روال getInstance() محدود کرده است .

الگوی تک برگ چه موقع استفاده می شود ؟

وقتی بخواهیم در برنامه از یک کلاس خاص تنها یک متغیر داشته باشیم.

کلاس با الگوی شمارشی با نوع محافظت شده (Enum)

برخی زبانها مانند ++C دارای ساختار دادهای به نام نوع شمارشی (Enumertion) هستند که از این پس آنها را شمارشی خواهیم خواند شمارشی ها در واقع فهرستی از ثوابت هستند اما این ثوابت محدود به شرایطی هستند مثلا نمی توانند رفتار خاصی را در نظر بگیرند یا افزودن ثوابت جدید به آنها دشوار است با تمام این اوصاف الگوی شمارشی راهی شی گرا برای تعریف ثوابت فراهم کرده است و به جای تعریف ثوابت صحیح (در ++C) برای هر نوع ثابت کلاسی تعریف می کنیم .

```
public final class Size{
   // statically define valid values of Size
    public static final Size  SMALL   =new Size('S');
    public static final Size  MEDIUM=new Size('M');
    public static final Size  BIG         =new Size('B');
    // helps to iterator over enum values
    public static final Size [] SIZE={ SMALL , MEDIUM , BIG};
    // instance variable for holding onto display value
    private final char display;
    // do not allow instantiation by outside objects
     public Size(char value){
       display=value
    }
    public char getValue {
           return display;
    }
    public String toString(){
           return new String (display);
    }
}
```

کلاس Size ساده است این کلاس از نوع final تعریف شده است لذا هیچ کلاسی از آن نمی توان مشتق شود و این کلاس گروهی از ثوابت را تعریف می کند که ثابت ها اختصاصی تعریف شده است که نمی توان به طور مستقیم به آنها دسترسی پیدا کرد در عوض دسترسی به ثوابت تعریف شده کلاس ممکن خواهد بود

Size.MEDIUM

منابع :

http://www.irandevelopers.com/
http://docs.sun.com

نویسنده :

محمد باقر معموری mamouri@ganjafzar.com
ویراستار و نویسنده قسمت های تکمیلی :
احسان شاه بختی zehs_sha@yahoo.com

کتاب :

Java در 21 روز انتشارات نص
برنامه نویسی شی گرا انتشارات نص

دستورات کنترلی

زبانهای برنامه نویسی از دستورات کنترلی استفاده می کنند تا جریان اجرای برنامه را پیشرفت داده و براساس تغییرات حالت یک برنامه شاخه هایی از آن برنامه منشعب نمایند . دستورات کنترلی برنامه در جاوا را می توان در طبقه بندی بعدی گنجاند : انتخاب (selection) ، تکرار (iteration)، و پرش (jump) . دستورات انتخاب به برنامه شما امکان میدهند تا مسیرهای متفاوت اجرای برنامه را براساس حاصل یک عبارت یا حالت خاص یک متغیر انتخاب نمایید . دستورات تکرار اجرای برنامه را قادر می سازد تا یک یا چند عبارت را تکرار نماید (یعنی دستورات تکرار حلقه ها را تشکیل می دهند). دستورات پرش به برنامه شما امکان می دهند تا یک روش اجرای غیر خطی داشته باشید . کلیه دستورات کنترلی جاوا را در اینجا بررسی نموده ایم . نکته : اگر C/ C++ را میدانید، دستورات کنترلی جاوا برای شما بسیار آشنا هستند . در حقیقت ، دستورات کنترلی جاوا برای برنامه نویسان C/ C ++ بسیار یکسان است . اما تفاوتهای محدودی وجود دارد بخصوص در دستورات break و......

دستورات انتخاب در جاوا

جاوا از دو دستور انتخاب پشتیبانی می کنند: if و switch . با این دستورات شما اجرای برنامه را براساس شرایطی که فقط حین اجرای برنامه اتفاق می افتند کنترل می کنید. اگر سابقه برنامه نویسی با C/C++ را ندارید، از قدرت و انعطاف پذیری موجود در این دو دستور متعجب و شگفت زده خواهید شد .

if

دستور if دستور انشعاب شرطی در جاوا است . از این دستور می توان استفاده نمود و اجرای برنامه را طی دو مسیر متفاوت به جریان انداخت . شکل کلی این دستور بصورت زیر است :

```
if( condition )statement 1;
else statement 2;
```

دراینجا هر statement ممکن است یک دستور منفرد یا یک دستور مرکب قرار گرفته در ابروها (یعنی یک بلوک) باشد . condition (شرط) هر عبارتی است که یک مقدار boolean را برمی گرداند . جمله else اختیاری است if . بصورت زیر کار می کند : اگر شرایط محقق باشد ، آنگاه statement 1 اجرا می شود . در غیر اینصورت (statement 2 در صورت وجود) اجرا خواهد شد . تحت هیچ شرایطی هر دو دستور با هم اجرا نخواهند شد . بعنوان مثال ، در نظر بگیرید :

```
int a, b;
//...
```

```
if(a < b )a = 0;
else b = 0;
```

در اینجا اگر a کوچکتر از b باشد ، آنگاه a برابر صفر می شود . در غیر اینصورت b برابر صفر قرار می گیرد . در هیچ شرایطی این دو متغیر در آن واحد برابر صفر نمی شوند . غالب اوقات ، عبارتی که برای کنترل if استفاده میشود شامل عملگرهای رابطه ای است . اما از نظر تکنیکی ضرورتی وجود ندارد . می توان با استفاده از یک متغیرboolean تکی ، ifرا همانطوریکه در بخش زیر مشاهده می کنید ، کنترل نمود .

```
boolean dataAvailable;
//...
if( dataAvailable)
ProcessData();
else
waitForMoreData();
```

بیاد آورید که فقط یک دستور می تواند مستقیما" بعدازif یاelse ا قرار گیرد .اگر بخواهید دستورات بیشتری داخل نمایید ، نیازی به ایجاد یک بلوک ندارید نظیر این قطعه که در زیر آمده است :

```
int bytesAvailable;
//...
if( bytesAvailable > 0 ){
ProcessData();
bytesAvailable- = n;
} else
waitForMoreData();
```

در اینجا ، هر دو دستور داخل بلوک if اجرا خواهند شد اگر bytes Available بزرگتر از صفر باشد . برخی از برنامه نویسان راحت ترند تا هنگام استفاده از if ، از ابروهای باز و بسته استفاده نمایند، حتی زمانیکه فقط یک دستور در هر جمله وجود داشته باشد .این امر سبب می شود تا بعدا "بتوان براحتی دستور دیگری را اضافه نمود و نگرانی از فراموش کردن ابروها نخواهید داشت . در حقیقت ، فراموش کردن تعریف یک بلوک هنگامی که نیاز است ، یکی از دلایل رایج بروز خطاها می باشد . بعنوان مثال قطعه زیر از یک کد را در نظر بگیرید :

```
int bytesAvailable;
//...
if( bytesAvailable > 0 ){
ProcessData();
bytesAvailable- = n;
```

```
} else
waitForMoreData();
bytesAvailable = n;
```

بنظر خیلی روشن است که دستور bytes Available=n طوری طراحی شده تا داخل جمله else اجرا گردد ، و این بخاطر سطح طراحی آن است . اما حتما" بیاد دارید که فضای خالی برای جاوا اهمیتی ندارد و راهی وجود ندارد که کامپایلر بفهمد چه مقصودی وجود دارد . این کد بدون مشکل کامپایل خواهد شد ، اما هنگام اجرا بطور ناصحیح اجرا خواهد شد . مثال بعدی داخل کدی که مشاهده می کنید تثبیت شده است :

```
int bytesAvailable;
//...
if( bytesAvailable > 0 ){
ProcessData();
bytesAvailable- = n;
} else {
waitForMoreData();
bytesAvailable = n;
}
```

if های تودرتو شده Nested ifs

یک nested if یک دستور if است که هدف if یا else ا دیگری باشد .if های تودرتو در برنامه نویسی بسیار رایج هستند. هنگامیکه if ها را تودرتو می کنید، مهمترین چیزی که باید بخاطر بسپارید این است که یک دستور elseهمیشه به نزدیکترین دستورif خود که داخل همان بلوک else است و قبلا" با یک else همراه نشده ، مراجعه خواهد نمود . مثالی را مشاهده نمایید :

```
if(i == 10 ){
if(j < 20 )a = b;
if(k > 100 )c = d; // this if is
else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

همانگونه که توضیحات نشان می دهند ، elseنهایی با (20چون داخل همان بلوک قرار ندارد (اگر چه نزدیکترین if بدون else است) . بجای آن ، elseنهایی با (i==10)if همراه می شود . else داخلی به (k>100)if ارجاع می کند ، زیرا نزدیکترین if در داخل همان بلوک است .

نردبان if-else-if

یک ساختار برنامه نویسی رایج براساس یک ترتیب از if های تودرتو شده یا نردبان if-else-if است . این ساختار بصورت زیر است :

```
if(condition)
statement;
else if(condition)

statement;
else if(condition)
statement;
.
.
.
else
statement;
```

دستورات if از بالا به پایین اجرا می شوند . مادامیکه یکی از شرایط کنترل کننده if صحیح باشد(true) ، دستور همراه با آن if اجرا می شود ، و بقیه نردبان رد خواهد شد . اگر هیچکدام از شرایط صحیح نباشند، آنگاه دستور else نهایی اجرا خواهد شد . else نهایی بعنوان شرط پیش فرض عمل می کند ، یعنی اگر کلیه شرایط دیگر صحیح نباشند ، آنگاه آخرین دستور else انجام خواهد شد . اگر else نهایی وجود نداشته باشد و سایر شرایط ناصحیح باشند ، آنگاه هیچ عملی انجام نخواهد گرفت .

در زیر ، برنامه ای را مشاهده می کنید که از نردبان if-else-if استفاده کرده تا تعیین کند که یک ماه مشخص در کدام فصل واقع شده است .

```
// Demonstrate if-else-if statement.
class IfElse {
public static void main(String args[] ){
int month = 4; // April
String season;

if(month == 12 || month == 1 || month == 2)
season = "Winter";
else if(month == 3 || month == 4 || month == 5)
season = "Spring";
else if(month == 6 || month == 7 || month == 8)
```

```
season = "Summer";
else if(month == 9 || month == 10 || month == 11)
season = "Autumn";
else
season = "Bogus Month";

System.out.println("April is in the"   season   ".");
}
}
```

خروجی این برنامه بقرار زیر می باشد :

```
April is in the Spring.
```

ممکن است بخواهید این برنامه را تجربه نمایید . خواهید دید که هیچ فرقی ندارد که چه مقداری به month بدهید ، یک و فقط یک دستور انتساب داخل نردبان اجرا خواهد شد .

switch

دستور switch ، دستور انشعاب چند راهه در جاوا است . این دستور راه ساده ای است برای تغییر مسیر اجرای بخشهای مختلف یک کد براساس مقدار یک عبارت . اینروش یک جایگزین مناسب تر برای مجموعه های بزرگتر از دستورات if-else-if است شکل کلی دستور switchبقرار زیر می باشد :

```
switch(expression){
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
.
.
.
case valueN:
```

```
// statement sequence
break;
default:
// default statement sequence
}
```

expression

می تواند هر نوع ساده ای را برگرداند ، هر یک از مقادیر (values) در دستورات case باید از نوع سازگار با عبارت باشند . هر یک از مقادیر case باید یک مقدار لفظی منحصر بفرد باشد یعنی باید یک ثابت ، نه متغیر ، باشد دو برابر سازی مقادیر case مجاز نیست دستور switchبشرح فوق عمل می کند : مقدار عبارت با هر یک از مقادیر لفظیدر دستورات case مقایسه می شوند. اگر تطابق پیدا شود ، کد سلسله ای تعقیب کنندهآن دستور case اجرا خواهد شد . اگر هیچیک از ثابت ها با مقدار عبارت تطابق نیابند ، آنگاه دستور پیش فرض (default)اجرا خواهد شد ، اما دستور default اختیاری است . اگر هیچیک از case ها تطابق نیابد و default وجود نداشته باشد آنگاه عمل اضافی دیگری انجام نخواهد شد از دستور break داخل دستورswitch استفاده شده تا سلسله یک دستور را پایان دهد .هنگامیکه با یک دستور break مواجه می شویم ، اجرا به خط اول برنامه که بعد از کل دستور switch قرار گرفته ، منشعب خواهد شد . این حالت تاثیر پریدنswitch است . در زیر مثال ساده ای را مشاهده می کنید که از دستورswitch استفاده نموده است :

```
// A simple example of the switch.
class SampleSwitch {
public static void main(String args[] ){
for(int i=0; i<6; i++  )
switch(i ){
case 0:
System.out.println("i is zero.");
break;
case 1:
System.out.println("i is one.");
break;
case 2:
System.out.println("i is two.");
break;
case 3:
System.out.println("i is three.");
break;
default:
System.out.println("i is greater then 3.");
}
}
}
```

خروجی این برنامه بقرار زیر می باشد :

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

همانطوریکه مشاهده می کنید ، داخل حلقه ، دستوراتی که همراه ثابت case بوده و با i مطابقت داشته باشند ، اجرا خواهند شد . سایر دستورات پشت سر گذاشته می شوند (bypassed) . بعد از اینکه i بزرگتر از 3 بشود ، هیچ دستور همراه case مطابقت نداشته ، بنابراین دستور پیش فرض (default) اجرا خواهد شد . دستور break اختیاری است . اگر break را حذف کنید ، اجرای برنامه با caseبعدی ادامه خواهد یافت . گاهی بهتر است چندین case بدون دستورات break در بین آنها داشته باشیم . بعنوان مثال ، برنامه بعدی را در نظر بگیرید :

```
// In a switch/ break statements are optional.
class MissingBreak {
public static void main(String args[] ){
for(int i=0; i<12; i++  )
switch(i ){
case 0:
case 1:
case 2:
case 3:
case 4:
System.out.println("i is less than 5");
break;
case 5:
case 6:
case 7:
case 8:
case 9:
System.out.println("i is less than 10");
break;
default:
System.out.println("i is 10 or more");
}
}
}
```

خروجی این برنامه بقرار زیر خواهد بود :

```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
```

```
i is 10 or more
i is 10 or more
```

همانطوریکه مشاهده می کنید، اجرا طی هر case ، بمحض رسیدن بـه یـک دسـتور break یـا انتهـای switch متوقـف مـی شـود . در حالیکه مثال قبلی برای توصیف نظر خاصی طراحی شده بود ، اما بهر حال حذف دستور break کاربردهای عملی زیادی در برنامه های واقعی دارد . برای نشان دادن کاربردهای واقعی تر این موضوع ، دوباره نویسی برنامه نمونه مربوط به فصول سال را مشاهده نمایید . این روایت جدیـد همان برنامه قبلی از switch استفاده می کند تا پیاده سازی موثرتری را ارائه دهد .

```
// An improved version of the season program.
class Switch {
public static void main(String args[] ){
int month = 4;
String season;
switch( month ){
case 12:
case 1:
case 2:
season = "Winter";
break;
case 3:
case 4:
case 5:
season = "Spring";
break;
case 6:
case 7:
case 8:
season = "Summer";
break;
case 9:
case 10:
case 11:
season = "Autumn";
break;
default:
```

```
season = "Bogus Month";
}
System.out.println("April is in the"   season   ".");

}
}
```

تودرتو کردن دستورات switch

می توانید از یک switch بعنوان بخشی از ترتیب یک دستور switch خارجی تر استفاده نمایید. این حالت را switch تودرتو مینامند. از آنجاییکه دستور switch تعریف کننده بلوک مربوط به خودش می باشد، هیچ تلاقی بین ثابتهای caseدر switch ر داخلی و آنهایی که در switch خارجی قرار گرفته اند ، بوجود نخواهد آمد . بعنوان مثال ، قطعه بعدی کاملا" معتبر است .

```
switch(count ){
case 1:
switch(target ){ // nested switch
case 0:
System.out.println("target is zero");
break;
case 1 :// no conflicts with outer switch
System.out.println("target is one");
break;
}
break;
case 2 ://...
```

در اینجا دستور1 case: درswitch ر داخلی با دستور1 case: در switch ر خارجی تلاقی نخواهد داشت . متغیر count فقط با فهرست case ها در سطح خارجی مقایسه می شود .اگر count برابر 1 باشد، آنگاه target با فهرست case های داخلی مقایسه خواهد شد .

بطور خلاصه ، سه جنبه مهم از دستور switch قابل توجه هستند : ۆ switchبا ifا متفاوت است چون switch فقط آزمایش کیفیت انجام می دهد ، در حالیکه if هر نوع عبارت بولی را ارزیابی می کند . یعنی که switch فقط بدنبال یک تطابق بین مقدار عبارت و یکی از ثابت های case خودش می گردد . ۆ دو ثابت case در switch ر مشابه نمی توانند مقادیر یکسان داشته باشند . البته ، یک دستور

switchقرار گرفته داخل یک switch خارجی تر می تواند ثابتهایcase مشترک داشته باشد . ؤ یک دستور switch معمولا" بسیار کاراتر از یک مجموعه از if های تودرتو شده است . آخرین نکته بخصوص جالب توجه است زیرا روشنگر نحوه کار کامپایلر جاوا می باشد . کامپایلر جاوا هنگامیکه یک دستور switch را کامپایل می کند ، به هر یک از ثابتهای case سرکشی نموده و یک جدول jump tableمی سازد که برای انتخاب مسیر اجرا براساس مقدار موجود در عبارت استفاده می شود . بنابراین ، اگر باید از میان گروه بزرگی از مقادیر انتخاب نمایید ، یک دستور switch نسبت به یک ترتیب از if-else ها که بطور معادل و منطقی کد بندی شده باشد ، بسیار سریعتر اجرا خواهد شد. کامپایلر قادر است اینکار را انجام دهد چون می داند که ثابتهایcase همه از یک نوع بوده و باید خیلی ساده با عبارت switchبرای کیفیت مقایسه شوند . کامپایلر چنین شناسایی را نسبت به یک فهرست طولانی از عبارات if ندارد .

دستورات تکرار iteration statements

دستورات تکرار در جاوا عبارتند از for ، while،وdo-while و . این دستورات آن چه را ما " حلقه " می نامیم ، ایجاد می کنند . احتمالا" می دانید که حلقه یک مجموعه از دستورالعملها را بطور تکراری اجرا می کند . تا اینکه یک شرط پایانی را ملاقات نماید . همانطوریکه بعدا " خواهید دید، جاوا حلقه ای دارد که برای کلیه نیازهای برنامه نویسی مناسب است .

while

حلقه while اساسی ترین دستور حلقه سازی (looping) در جاوا است . این دستور مادامیکه عبارت کنترل کننده ، صحیح (true) باشد، یک دستور یا یک بلوک را تکرار می کند . شکل کلی این دستور بقرار زیر است :

```
while(condition ){
// body of loop
}
```

شرط یا condition ممکن است هر عبارت بولی باشد . مادامیکه عبارت شرطی صحت داشته باشد ، بدنه حلقه اجرا خواهد شد . هنگامیکه شرط صحت نداشته باشد ، کنترل بلافاصله به خط بعدی کدی که بلافاصله پس از حلقه جاری قرار دارد ، منتقل خواهد شد . اگر فقط یک دستور منفرد در حال تکرار باشد ، استفاده از ابروها غیر ضروری است . در اینجا یک حلقه while وجود دارد که تا 10 را محاسبه کرده و دقیقا" ده خط"tick" را چاپ می کند .

```
// Demonstrate the while loop.
class While {
public static void main(String args[] ){
int n = 10;

while(n > 0 ){
```

```
System.out.println("tick"   n);
n--;
}
}
}
```

هنگامیکه این برنامه را اجرا می کنید، ده مرتبه "tick" را انجام خواهد داد :

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

از آنجاییکه حلقه while عبارت شرطی خود را در بالای حلقه ارزیابی میکند، اگر شرط ابتدایی ناصحیح باشد ، بدنه حلقه اجرا نخواهد شد .

بعنوان مثال ، در قطعه زیر ، فراخوانی println() هرگز اجرا نخواهد شد .

```
int a = 10, b = 20;

while(a < b)
System.out.println("This will not be displayed");
```

بدنه (while یا هر حلقه دیگر در جاوا) ممکن است تهی باشد. زیرا دستور تهی (دستوری که فقط شامل ; باشد) از نظر قواعد ترکیبی در جاوا معتبراست . بعنوان مثال ، برنامه زیر را در نظر بگیرید :

```
// The target of a loop can be empty.
class NoBody {
public static void main(String args[] ){
int i, j;

i = 100;
j = 200;
```

```
// find midpoint between i and j
while(  i <-- j); // no body in this loop

System.out.println("Midpoint is"  + i);
}
}
```

این برنامه نقطه میانی (midpoint) بین i و j را پیدا می‌کند و خروجی زیر را تولید خواهد کرد :

Midpoint is 150

در اینجا چگونگی کار حلقه while را می بینید . مقدار i افزایش و مقدار j کاهش می یابد . سپس این دو مقدار با یکدیگر مقایسه می شوند . اگر مقدار جدید i همچنان کمتر از مقدار جدید j باشد ، آنگاه حلقه تکرار خواهد شد . اگر i مساوی با یا بزرگتر از j بشود ، حلقه متوقف خواهد شد . تا هنگام خروج از حلقه ، iمقداری را می گیرد که بین مقادیر اولیهi وj و می باشد . (بدیهی است که این رویه هنگامی کار می کند که i کوچکتر از مقدار اولیه j باشد .) همانطوریکه می بینید ، نیازی به بدنه حلقه نیست ، کلیه عملیات داخل خود عبارت شرطی اتفاق می افتد . در کدهای حرفه ای نوشته شده دیگر جاوا ، وقتی که عبارت کنترل کننده توانایی مدیریت کلیه جزئیات خود را داشته باشد ، حلقه های کوتاه غالبا" بدون بدنه کد بندی می شوند .

do-while

گفتیم اگر عبارت شرطی کنترل کننده یک حلقه while در ابتدا ناصحیح باشد آنگاه بدنه حلقه اصلا" اجرا نمی شود . اما گاهی مایلیم در چنین شرایطی ، بدنه حلقه حداقل یکبار اجرا شود . بعبارت دیگر، در حالات خاصی مایلید تا عبارت پایان دهنده در انتهای حلقه را آزمایش کنید .خوشبختانه ، جاوا حلقه ای را عرضه می کند که دقیقا" همین کار را انجام می دهد . do-while : حلقه do-while همواره حداقل یکبار بدنه خود را اجرا می کند، زیرا عبارت شرطی آن در انتهای حلقه قرار گرفته است . شکل کلی آن بصورت زیر است :

```
do{
// body of loop
} while(condition);
```

هر تکرار از حلقه do-while ابتدا بدنه حلقه را اجرا نموده ، سپس به ارزیابی عبارت شرطی خود می پردازد . اگر این عبارت صحیح (true)باشد ، حلقه اجرا خواهد شد . در غیر اینصورت حلقه پایان می گیرد . نظیر کلیه حلقه های جاوا ، شرط باید یک عبارت بولی باشد .

اینجا یک روایت دیگر از برنامه (tick) وجود دارد که حلقه do-while را نشان می دهد . خروجی این برنامه مشابه برنامه قبلی خواهد بود :

```
// Demonstrate the do-while loop.
class DoWhile {
public static void main(String args[] ){
int n = 10;

do {
System.out.println("tick"   n);
n--;
} while(n > 0);
}
}
```

حلقه موجود در برنامه قبلی ، اگر چه از نظر تکنیکی صحیح است ، اما می توان آن را به شکل کاراتری بصورت زیر دوباره نویسی نمود :

```
do {
System.out.println("tick "   n);
} while--(n > 0);
```

در این مثال ، عبارت (0 <n) عمل کاهش n و آزمایش برای صفر را در یک عبارت گنجانده است . عملکرد آن بقرار بعدی است . ابتدا دستور n اجرا می شود و n را کاهش داده و مقدار جدید را به n برمی گرداند . این مقدار سپس با صفر مقایسه می شود . اگر بزرگتر از صفر باشد ، حلقه ادامه می یابد . در غیر اینصورت حلقه پایان می گیرد. حلقه do-while بویژه هنگام پردازش انتخاب منو بسیار سودمند است ، زیرا معمولا" مایلید تا بدنه یک حلقه منو حداقل یکبار اجرا شود . برنامه بعدی را که یک سیستم Help ساده را برای دستورات تکرار و انتخاب در جاوا پیاده سازی می کند در نظر بگیرید :

```
// Using a do-while to process a menu selection -- a simple help system.
class Menu {
public static void main(String args[])
throws java.io.IOException {
char choice;

do {
System.out.prinln("Help on:");
System.out.prinln(" 1 .if");
System.out.prinln(" 2 .switch");
System.out.prinln(" 3 .while");
```

```
System.out.prinln(" 4 .do-while");
System.out.prinln(" 5 .for\n");
System.out.prinln("Choose one:");
choice =( char )System.in.read();
} while(choice < '1' || choice > '5');

System.out.println("\n");
switch(choice ){
case '1':
System.out.println("The if:\n");
System.out.println("if(condition )statement;");
System.out.println("else statement;");
break;
case '2':

System.out.println("The switch:\n");
System.out.println("switch(expression ){");
System.out.println(" case constant:");
System.out.println(" statement sequence");
System.out.println(" break;");
System.out.println(" //... ");
System.out.println("}");
break;
case '3':
System.out.println("The switch:\n");
System.out.println(while(condition )statement;");
break;
case '4':
System.out.println("The do-while:\n");
System.out.println("do {");
System.out.println(" statement;");
System.out.println("} while( condition);");
break;
case '5':
System.out.println("The for:\n");
System.out.print("for(init; condition; iteration)");
System.out.println(" statement;");
break;
```

```
}
}
}
```

مشاهده می کنید اکنون یک اجرای نمونه تولید شده توسط این برنامه را :

```
Help on:
1 .if
2 .switch
3 .while
4 .do-while
5 .for
Choos one:
4
```

The do-while:

```
do {
statement;
} while( condition);
```

در برنامه ، از حلقه do-while برای تصدیق اینکه کاربر یک گزینه معتبر را وارد کرده باشد ، استفاده می شود . در غیر اینصورت ، به کاربر مجددا" اعلان خواهد شد . از آنجاییکه منو باید حداقل یکبار بنمایش درآید ، do-whileحلقه کاملی برای انجام این مقصود است .

چند نکته دیگر درباره این مثال : دقت کنید که کاراکترهااز صفحه کلید بوسیله فراخوانی system.in.read() خوانده می شوند . این یکی از توابع ورودی کنسول در جاوا است .

اگر چه بررسی تفصیلی روشهای I/o جاوا به بحثهای بعدی موکول شده ، اما از system.in.read() در اینجا برای بدست آوردن گزینه کاربر استفاده شده است . این تابع کاراکترها را از ورودی استاندارد می خواند (که بعنوان عدد صحیح برگردان شد ، این دلیلی است که چرا مقدار برگردان از طریق تبدیل (cast) به char تبدیل شده است). بصورت پیش فرض ، ورودی استاندارد، بافر شده خطی است (line buffered)بنابراین قبل از اینکه کاراکترهایی را که تایپ کرده اید به برنامه اتان ارسال کنید ، باید کلید ENTER را فشار دهید . (این حالت مشابه C /C است و احتمالا" از قبل با آن آشنایی دارید .) ورودی کنسول در جاوا کاملا" محدود شده و کار با آن بسیار مشکل است . بعلاوه اکثر برنامه و ریز برنامه های واقعی نوشته شده با جاوا گرافیکی و پنجره ای هستند. از سوی دیگر : چون از

system.in.read()استفاده شده ، برنامه باید جمله throwsjava.io.IoException را کاملا" توصیف نماید . این خط برای مدیریت خطاهای ورودی ضروری است . این بخشی از جنبه های مختلف اداره استثنائ در جاوا است که بعدا" بررسی خواهد شد .

for

خواهید دید که حلقه for یک ساختار قدرتمند و بسیار روان است .شکل کلی دستور for بصورت زیر است :

```
for(initialization; condition; iteration; ){
// body
}
```

اگر فقط یک دستور باید تکرار شود ، نیازی به ابروها نیست . عملکرد حلقه for بشرح بعدی است . وقتی که حلقه برای اولین بار شروع می شود بخض مقدار دهی اولیه در حلقه اجرا می شود . معمولا" ، این بخش یک عبارت است که مقدار متغیر کنترل حلقه را تعیین می کند ، که بعنوان یک شمارشگر ، کنترل حلقه را انجام خواهد داد . مهم است بدانیم که عبارت مقدار دهی اولیه فقط یکبار اجرا می شود . سپس شرط مورد ارزیابی قرار می گیرد . این شرط باید یک عبارت بولی باشد . این بخش معمولا" مقدار متغیر کنترل حلقه را با مقدار هدف مقایسه می کند .اگر عبارت صحیح (true) باشد، آنگاه بدنه حلقه اجرا خواهد شد . اگر ناصحیح باشد حلقه پایان می گیرد . بعد، بخش تکرار (iteration)حلقه اجرا می شود . این بخش معمولا" عبارتی است که مقدار متغیر کنترل را افزایش یا کاهش می دهد. آنگاه حلقه تکرار خواهد شد ، ابتدا عبارت شرطی را ارزیابی می کند ، سپس بدنه حلقه را اجرا می کند و سرانجام عبارت تکرار را در هر گذر (pass) اجرا میکند. این روال آنقدر دادمه می یابد تا عبارت شرطی ناصحیح (false) گردد . در زیر روایت جدیدی از برنامه "tick" را می بینید که از یک حلقه for استفاده کرده است :

```
// Demonstrate the for loop.
 class ForTick {
 public static void main(String args[] ){
 int n;
 for(n=10; n>0; n--)
 System.out.println("tick" +  n);
 }
 }
```

اعلان متغیرهای کنترل حلقه داخل حلقه for

غالبا" متغیری که یک حلقه for را کنترل می کند ، فقط برای همان حلقه مورد نیاز بوده و کاربری دیگری ندارد . در چنین حالتی ، می توان آن متغیر را داخل بخش مقدار دهی اولیه حلقه for اعلان نمود . بعنوان مثال در اینجا همان برنامه قبلی را مشاهده می کنید که متغیر کنترل حلقه یعنی n بعنوان یک int در داخل حلقه for اعلان شده است .

```
// Declare a loop control variable inside the for.
class ForTick {
public static void main(String args[] ){

// here/ n is declared inside of the for loop
for(int n=10; n>0; n--)
System.out.println("tick"   n);
}
}
```

هنگامیکه یک متغیر را داخل یک حلقه for اعلان می کنید ، یک نکته مهم را باید بیاد داشته باشید : قلمرو آن متغیر هنگامیکه دستور for انجام می شود ، پایان می یابد . (یعنی قلمرو متغیر محدود به حلقه for است .) خارج از حلقه for حیات آن متغیر متوقف می شود . اگر بخواهید از این متغیر کنترل حلقه در جای دیگری از برنامه اتان استفاده کنید ، نباید آن متغیر را داخل حلقه for اعلان نمایید . درشرایطی که متغیر کنترل حلقه جای دیگری موردنیاز نباشد، اکثر برنامه نویسان جاوا آن متغیر را داخل for اعلان می کنند . بعنوان مثال ، در اینجا یک برنامه ساده را مشاهده می کنید که بدنبال اعداد اول می گردد. دقت کنید که متغیر کنترل حلقه ، چون جای دیگری مورد نیاز نیست ، داخل forاعلان شده است .

```
// Test for primes.
class FindPrime {
public static void main(String args[] ){
int num;
boolean isPrime = true;

num = 14;
for(int i=2; i < num/2; i  ++ ){
if((num % i )== 0 ){
isPrime = false;
break;
}
```

```
}
if(isPrime )System.out.println("Prime");
else System.out.println("Not Prime");
}
}
```

استفاده از کاما Comma

شرایطی پیش می آید که مایلید بیش از یک دستور در بخش مقدار دهی اولیه (initalization) و تکرار (iteration) بگنجانید . بعنوان مثال ، حلقه موجود در برنامه بعدی را در نظر بگیرید :

```
Class Sample {
public static void main(String args[] ){
int a, b;
 b = 4;
for(a=1; a+ System.out.println("a = " + a);
System.out.println("b = " + b);
b--;
}
}
}
```

همانطوریکه می بینید ، حلقه توسط ارتباط متقابل دو متغیر کنترل می شود . از آنجاییکه حلقه توسط دو متغیر اداره می شود ، بجای اینکه b را بصورت دستی اداره کنیم ، بهتر است تا هر دو را در دستور for بگنجانیم . خوشبختانه جاوا راهی برای اینکار دارد . برای اینکه دو یا چند متغیر بتوانند یک حلقه for را کنترل کنند ، جاوا به شما امکان می دهد تا چندین دستور را در بخشهای مقدار دهی اولیه و تکرار حلقه for قرار دهید . هر دستور را بوسیله یک کاما از دستور بعدی جدا می کنیم . حلقه for قبلی را با استفاده از کاما ، خیلی کاراتر از قبل می توان بصورت زیر کد بندی نمود :

```
// Using the comma.
class Comma {
public static void main(String args[] ){
int a, b;

for(a=1; b=4; a++)
System.out.println("a = "   a);
System.out.println("b = "   b);
```

```
}
}
}
```

در این مثال ، بخش مقدار دهی اولیه ، مقادیر a و b را تعیین می کند . هربار که حلقه تکرار می شود ، دو دستور جدا شده توسط کاما در بخش تکرار (iteration) اجرا خواهند شد . خروجی این برنامه بقرار زیر می باشد :

```
a=1
b=4
a=2
b=3
```

نکته : اگر با C/ C++ آشنایی دارید ، حتما" می دانید که در این زبانها ، علامت کاما یک عملگر است که در هر عبارت معتبری قابل استفاده است . اما در جاوا اینطور نیست . در جاوا ، علامت کاما یک جدا کننده است که فقط در حلقه for قابل اعمال می باشد .

برخی گوناگونیهای حلقه for

حلقه for از تعدادی گوناگونیها پشتیبانی می کند که قدرت و کاربری آن را افزایش می دهند . دلیل انعطاف پذیری آن است که لزومی ندارد که سه بخش مقداردهی اولیه ، آزمون شرط و تکرار ، فقط برای همان اهداف مورد استفاده قرار گیرند . در حقیقت ، سه بخش حلقه forبرای هر هدف مورد نظر شما قابل استفاده هستند . به چند مثال توجه فرمائید . یکی از رایجترین گوناگونیها مربوط به عبارت شرطی است . بطور مشخص ، لزومی ندارد این عبارت ، متغیر کنترل حلقه را با برخی مقادیر هدف آزمایش نماید . در حقیقت ، شرط کنترل کننده حلقه for ممکن است هر نوع عبارت بولی باشد . بعنوان مثال ، قطعه زیر را در نظر بگیرید :

```
boolean done = false;

for(int i=1; !done; i   ){
//...
if(intettupted )()done = true;
}
```

در این مثال ، حلقه for تا زمانیکه متغیر بولی done معادل true بشود ، اجرا را ادامه خواهد داد . این مثال مقدار i را بررسی نمی کند .

اکنون یکی دیگر از گوناگونیهای جالب حلقه for را مشاهده می کنید. ممکن است یکی یا هر دو عبارت مقدار دهی اولیه و تکرار غایت باشند ، نظیر برنامه بعدی :

```
// Parts of the for loop can be empty.
```

```
class ForVar {
public static void main(String args[] ){
int i;
boolean done = false;

i = 0;
for (; !done; ) {
System.out.println("i is"   i);
if(i == 10 )done = true;
i  ;
}
}
}
```

در اینجا عبارتهای مقدار دهی اولیه و تکرار به خارج از forانتقال یافته اند .برخی از بخشهای حلقه for تهی هستند . اگر چه در این مثال ساده چنین حالتی هیچ ارزشی ندارد ، اما در حقیقت شرایطی وجود دارد که این روش بسیار کارا و سودمند

خواهد بود. بعنوان مثال ، اگر شرط اولیه بصورت یک عبارت پیچیده و در جای دیگری از برنامه قرار گرفته باشد و یا تغییرات متغیر کنترل حلقه بصورت غیر ترتیبی و توسط اعمال اتفاق افتاده در داخل بدنه حلقه تعیین شود ، پس بهتر است که این بخشها را در حلقه for تهی بگذاریم . اکنون یکی دیگر از گوناگونیهای حلقه for را مشاهده می کنید. اگر هر سه بخش حلقه for را تهی بگذارید ، آنگاه بعمد یک حلقه نامحدود (حلقه ای که هرگز پایان نمی گیرد) ایجاد کرده اید . بعنوان مثال :

```
for (; ; ) {
//...
}
```

این حلقه تا ابد ادامه خواهد یافت ، زیرا هیچ شرطی برای پایان گرفتن آن تعبیه نشده است . اگر چه برخی برنامه ها نظیر پردازشهای فرمان سیستم عامل مستلزم یک حلقه نامحدود هستند ، اما اکثر حلقه های نامحدود در واقع حلقه هایی هستند که ملزومات پایان گیری ویژه ای دارند . بزودی خواهید دید ، راهی برای پایان دادن به یک حلقه حتی یک حلقه نامحدود نظیر مثال قبلی وجود دارد که از عبارت شرطی معمولی حلقه استفاده نمی کند .

حلقه های تودرتو

نظیر کلیه زبانهای برنامه نویسی ، جاوا نیز امکان تودرتو کردن حلقه ها را دارد . یعنی یک حلقه داخل حلقه دیگری قرار خواهد گرفت . بعنوان مثال ، در برنامه بعدی حلقه های for تودرتو نشده اند :

```
// Loops may be nested.
class Nested {
public static void main(String args[] ){
int i/ j;

for(i=0; i<10; i++   ){
for(j=i; j<10; j++  )
System.out.print(".");
System.out.println();
}
}
}
```

خروجی تولید شده توسط این برنامه بقرار زیر می باشد .... :

....

دو دستور کنترلی

این دو دستور یک جنبه بسیار مهم از جاوا یعنی بلوک های کد (block of code) را تشریح می کنند .

دستور if(if statement()

دستور if در جاوا نظیر دستور If درهر یک از زبانهای برنامه نویسی کار میکند .بعلاوه این دستور از نظر قواعد صرف و نحو با دستورif در Cو C++ و شباهت دارد . ساده ترین شکل آن را در زیر مشاهده می کنید :

```
if( condition )statement;
```

در اینجا شرط (condition) یک عبارت بولی (Boolean) است . اگر شرایط درست باشد ، آنگاه دستور اجرا خواهدشد . اگر شرایط صحیح نباشد ، آنگاه دستور پشت سر گذاشته می شود .(bypassed) بعنوان مثال در نظر بگیرید :

```
if(num < 100 )println("num is less then 100");
```

در این حالت ، اگر متغیر num شامل مقداری کوچکتر از 100 باشد ، عبارت شرطی درست بوده و println() اجرا خواهد شد . اگر numشامل مقداری بزرگتر یا مساوی 100باشد ، آنگاه روش println() پشت سر گذاشته می شود . بعدا" خواهید دید که جاوا یک ضمیمه کامل از عملگرهای رابطه ای (Relational) تعریف می کند که قابل استفاده در عبارات شرطی هستند . چند نمونه از آنها بشرح زیر است :

عملگر	مفهوم آن
<	کوچکتر از
>	بزرگتر از
= =	مساوی با

دقت داشته باشید که آزمایش تساوی با علامت تساوی انجام می گیرد . در زیر برنامه ای مشاهده می کنید که یک دستور if را توصیف کرده است :

```
/*
Demonstrate the if.
Call this file "IfSample.java".
*/
class IfSample {
public static void main( String args [] ){
int x/ y;

x = 10;
y = 20;
if(x < y )System.out.println("x is less than y");

x = x * 2;
if(x == y )System.out.println("x now equal to y");

x = x * 2;
if(x > y )System.out.println("x now greater than y");
```

```
// this won't display anything
if(x == y )System.out.println("you won't see this");
}
}
```

خروجی تولید شده توسط این برنامه بشرح زیر خواهد بود :

```
x is less than y
x now equal to y
x now greater than y
```

به یک نکته دیگر در این برنامه دقت نمایید . خط

```
int x, y;
```

دو متغیر x و y را با استفاده از فهرست جدا شده با کاما اعلان می کند .

حلقه for

شاید از تجربیات قبلی در برنامه نویسی تا بحال فهمیده باشید که دستورات حلقه (loop statements) یک بخش بسیار مهم در کلیه زبانهای برنامه نویسی هستند. جاوا نیز از این قاعده مستثنی نیست . در حقیقت همانگونه که در فصل پنجم خواهید دید ، جاوا یک دسته بندی پر قدرت از ساختارهای حلقه ای عرضه می کند . شاید از همه این ساختارها سلیس تر حلقه for باشد . اگر با C و C++ و آشنایی داشته باشید خوشحال خواهید شد که بدانید نحوه کار حلقه های for در جاوا با این زبانها مشابه است . اگر با C و ++ و C آشنایی ندارید ، باز هم فراگیری استفاده از حلقه for بسیار ساده خواهد بود . ساده ترین شکل این حلقه بقرار زیر می باشد :

```
for( initialization; condition )statement;
```

دستور اجرای مکرر شرایط مقداردهی اولیه بخش مقداردهی اولیه در یک حلقه در معمول ترین شکل خود یک مقدار اولیه را در یک متغیر کنترل اگر ماحصل این آزمایش صحیح باشد ، حلقه for به تکرار خود ادامه می دهد . اگر حاصل ناصحیح باشد ، حلقه متوقف خواهد شد . عبارت اجرای مکرر تعیین کننده این است که متغیر کنترل حلقه پس از هر بار تکرار حلقه چگونه تغییر خواهد کرد . برنامه کوتاه زیر توصیف کننده یک حلقه for می باشد :

```
/*
Demonstrate the for loop.
Call this file "ForTest.java".
*/
```

```
class ForTest {
public static void main(String args [] ){
int x;

for(x = 0; x<10; x = x 1)
System.out.println("This is x :"   x);
}
}
```

این برنامه ، خروجی زیر را تولید خواهد نمود :

```
this is x:0
this is x:1
this is x:2
this is x:3
this is x:4
this is x:5
this is x:6
this is x:7
this is x:8
this is x:9
```

در این مثال ، xمتغیر کنترل حلقه است . در بخش مقداردهی اولیه حلقه for به ایـن متغیـر مقـدار صـفر داده مـی شـود. در شـروع هـر تکـرار (شامل مرحله اول) آزمایش شرط 10اجرا خواهد شد ، و سپس بخش اجرای مکرر حلقـه اجـرا خواهـد شـد . ایـن روال مادامیکـه آزمـایش شرایط صحیح باشد ، ادامه می یابد. بعنوان یک نکته قابل تامل ،در برنامه های حرفه ای نوشته شده توسط جـاوا بنـدرت بخـش اجـرای مکـرر حلقه بصورت برنامه قبلی نوشته می شود . یعنی شما بندرت دستوری مثل عبارت زیر خواهید دید :

```
x = x   1;
```

دلیل این است که جاوا دربرگیرنده یک عملگر افزایشی ویژه است که همین عملیات را بطور موثرتری انجام مـی دهـد . ایـن عملگـر افزایشـی است (دو علامت جمع در کنار هم) . عملگر افزایشی ، عملوند خود را یکی یکی افزایش خواهد داد . بـا اسـتفاده از عملگـر افزایشـی دسـتور قبلی را می توان بصورت زیر نوشت :

```
x  ;
```

بدین ترتیب for در برنامه قبلی را معمولا "بصورت زیر می نویسند :

```
for(x = 0; x<10; x++  )
```

ممکن است بخواهید این مورد را آزمایش کنید . همانطوریکه خواهید دید ، حلقه درست مثل قبل و بهمان ترتیب اجرا خواهد شد . جاوا همچنین یک عملگر کاهشی (decrement) فراهم نموده که باعلامت (– –) مشخص یکی یکی کاهش خواهد داد.

آرایه ها

یک آرایه گروهی از متغیرهای یک نوع است که با یک نام مشترک به آنها ارجاع می شود . می توان آرایه ها را برای هر یـک از انـواع ایجاد نمود و ممکن است این آرایه ها دارای یک یا چندین بعد باشند . برای دسترسی به یک عضو آرایه از نمایـه (index) آن آرایـه استفاده می شود . آرایه ها یک وسیله مناسب برای گروه بندی اطلاعات مرتبط با هم هستند . نکته : اگر با C و C++ و آشنایی داریـد ، آگاه باشید . آرایه ها در جاوا بطور متفاوتی نسبت به زبانهای دیگر کار می کنند .

آرایه های یک بعدی

آرایه یک بعدی بطور ضروری فهرستی از متغیرهای یکنوع است . برای ایجاد یک آرایه ، باید یـک متغیـر آرایـه از نـوع مـورد نظرتـان ایجاد کنید . فرم عمومی اعلان یک آرایه یک بعدی بقرار زیر است :

```
type var-name [];
```

نام متغیر نوع

در اینجا type اعلان کننده نوع اصلی آرایه است . نوع اصلی تعیین کننده نوع داده برای هـر یـک از اعضـائ داخـل در آرایـه اسـت . بنابراین ، نوع اصلی آرایه تعیین می کند که آرایه چـه نـوعی از داده را نگهـداری مـی کنـد . بعنـوان مثـال ، در زیـر یـک آرایـه بـا نـام month-days با نوع آرایه ای از عدد صحیح اعلان شده است .

```
int month_days[];
```

اگر چه این اعلان تثبیت می کند که month-days یک متغیر آرایه است ، اما بطور واقعی آرایه ای وجود ندارد . در حقیقت ، مقدار month-days برابر تهی (null) می باشد که یک آرایه بدون مقدار را معرفی می کند . برای پیوند دادن month-days با یک آرایه واقعی و فیزیکی از اعداد صحیح ، باید از یک عملگر new استفاده نموده و به month-days منتسب کنید . new یک عملگراست که حافظه را اختصاص میدهد .بعدا " newرا با دقت بیشتری بررسی می کنیم ، اما لازم است که هم اکنون از آن استفاده نموده و حافظه را برای آرایه ها تخصیص دهید . فرم عمومی new آنگونه که برای آرایه های یک بعدی بکار می رود بقرار زیر ظاهر خواهد شد :

```
array-var=new type [size];
```

اندازه نوع متغیر آرایه

در اینجا type مشخص کننده نوع داده ای است کـه تخصـیص داده مـی شـود، sizeمشـخص کننـده تعـداد اعضـائ آرایـه اسـت و array-varمتغیر آرایه است که به آرایه پیوند می یابد . یعنی برای اسـتفاده از new در تخصـیص یـک آرایـه ، بایـد نـوع و تعـداد اعضایی که تخصیص می یابند را مشخص نمایید . اعضائ آرایه که توسط new تخصیص می یابند بطور خودکار با مقدار صـفر مقـدار دهی اولیه می شوند . این مثال یک آرایه 12 عضوی از اعداد صحیح را تخصیص داده و آنها را به month-days پیوند می دهد .

```
month_days = new int[12];
```

بعد از اجرای این دستور ، month-daysبه یک آرایه 12 تایی از اعداد صحیح ارجاع خواهد نمود . بعلاوه کلیه اجـزائ در آرایـه با عدد صفر مقدار دهی اولیه خواهند شد . اجازه دهید مرور کنیم : بدست آوردن یک آرایـه مسـتلزم پـردازش دو مرحلـه ای اسـت . اول باید یک متغیر با نوع آرایه مورد نظرتان اعلان کنید . دوم باید حافظـه ای کـه آرایـه را نگهـداری مـی کنـد ، بـا اسـتفاده از new تخصیص دهید و آن را به متغیر آرایه نسبت دهید . بنابراین در جاوا کلیه آرایه ها بطور پویا تخصیص مـی یابنـد . اگـر مفهـوم تخصـیص پویا برای شما ناآشناست نگران نباشید . این مفهوم را بعدا "تشریح خواهیم کرد . هر بار که یک آرایه را تخصیص می دهید ، می توانید بوسیله مشخص نمودن نمایه آن داخل کروشه [] به یک عضو مشخص در آرایه دسترسی پیدا کنید . کلیه نمایه های آرایه ها با عدد صفر شروع می شوند. بعنوان مثال این دستور مقدار 28 را به دومین عضو month-days نسبت می دهد .

```
month_days[1] = 28;
```

خط بعدی مقدار ذخیره شده در نمایه 3 را نمایش می دهد .

```
System.out.println(month_days[3]);
```

با کنار هم قرار دادن کلیه قطعات ، در اینجا برنامه ای خواهیم داشت که یک آرایه برای تعداد روزهای هر ماه ایجاد می کند .

```
// Demonstrate a one-dimensional array.
class Array {
public static void main(String args[] ){
int month_days[];
month_days = new int[12];
```

```
month_days [0] = 31;
month_days [1] = 28;
month_days [2] = 31;
month_days [3] = 30;
month_days [4] = 31;
month_days [5] = 30;
month_days [6] = 31;
month_days [7] = 31;
month_days [8] = 30;
month_days [9] = 31;
month_days [10] = 30;
month_days [11] = 31;
System.out.println("April has " + month_days[3] + " days .");
}
}
```

وقتی این برنامه را اجرا میکنید ، برنامه ، تعداد روزهای ماه آوریل را چاپ میکند. همانطوریکه ذکر شد، نمایه های آرایه جاوا با صفر شروع می شوند، بنابراین تعداد روزهای ماه آوریل در month-days[3] برابر 30 می باشد . این امکان وجود دارد که اعلان متغیر آرایه را با تخصیص خود آرایه بصورت زیر ترکیب نمود

```
int month_days[] = new int[12];
```

این همان روشی است که معمولا" در برنامه های حرفه ای نوشته شده با جاوا مشاهده می کنید . می توان آرایه ها را زمان اعلانشان ، مقدار دهی اولیه نمود . پردازش آن بسیار مشابه پردازشی است که برای مقدار دهی اولیه انواع ساده استفاده می شود . یک مقدار ده اولیه آرایه فهرستی از عبارات جدا شده بوسیله کاما و محصور شده بین ابروهای باز و بسته می باشد . کاماها مقادیر اجزائ آرایه را از یکدیگر جدا می کنند . آرایه بطور خودکار آنقدر بزرگ ایجاد می شود تا بتواند ارقام اجزایی را که در مقدار ده اولیه آرایه مشخص کرده اید ، دربرگیرد . نیازی به استفاده از new وجود ندارد . بعنوان مثال ، برای ذخیره نمودن تعداد روزهای هر ماه ، کد بعدی یک آرایه مقدار دهی اولیه شده از اعداد صحیح را بوجود می آورد :

```
// An improved version of the previous program.
class AutoArray {
public static void main(String args[] ){
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,31 };
System.out.println("April has " + month_days[3] + " days .");
```

```
}
}
```

وقتی این برنامه را اجرا کنید ، همان خروجی برنامه قبلی را خواهید دید . جاوا بشدت کنترل می کند تا مطمئن شود که بطور تصادفی تلاشی برای ذخیره نمودن یا ارجاع مقادیری خارج از دامنه آرایه انجام ندهید . سیستم حین اجرای جاوا کنترل می کند که کلیه نمایه های آرایه ها در دامنه صحیح قرار داشته باشند . (از این نظر جاوا کاملا"با C او C++ و متفاوت است که هیچ کنترل محدوده ای در حین اجرا انجام نمی دهند .) بعنوان مثال ، سیستم حین اجرا ، مقدار هر یک از نمایه ها به month-days را کنترل می کند تا مطمئن شود که بین ارقام 0 و 11 داخل قرار داشته باشند . اگر تلاش کنید تا به اجزائ خارج از دامنه آرایه (اعداد منفی یا اعدادی بزرگتر از طول آرایه) دسترسی یابید، یک خطای حین اجرا (run-time error) تولید خواهد شد . در زیر یک مثال پیچیده تر مشاهده می کنید که از یک آرایه یک بعدی استفاده می کند . این برنامه میانگین یک مجموعه از ارقام را بدست می آورد .

```
// Average an array of values.
class Average {
public static void main(String args[] ){
double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
double result = 0;
int i;
for(i=0; i<5; i++)
result = result + nums[i];
System.out.println("Average is " + result / 5);
}
}
```

آرایه های چند بعدی

در جاوا آرایه های چند بعدی در واقع آرایه ای از آرایه ها هستند . این قضیه همانطوریکه انتظار دارید ظاهر و عملکردی مشابه آرایه های چندبعدی منظم (regular) دارد . اما خواهید دید که تاوتهای ظریفی هم وجود دارند . برای اعلان یک متغیر آرایه چند بعدی ، با استفاده از مجموعه دیگری از کروشه ها هر یک از نمایه های اضافی را مشخص می کنید. بعنوان مثال ، عبارت زیرر یک متغیر آرایه دو بعدی بنامtwoD را اعلان می کند .

```
int twoD[][] = new int[4][5];
```

این عبارت یک آرایه 4در 5ر را تخصیص داده و آن را به twoD نسبت می دهد . از نظر داخلی این ماتریس بعنوان یک آرایه از آرایه نوع int پیاده سازی خواهد شد . بطور فرضی ، این آرایه را می توان بصورت شکل زیر نمایش داد .

Right index determines column.

|| || || || ||
V V V V V

[0][4] | [0][3] | [0][2] | [0][1] | [0][0] >

[1][4] | [1][3] | [1][2] | [1][1] | [1][0] >
Left index
determines
[2][4] | [2][3] | [2][2] | [2][1] | [2][0] .> row

[3][4] | [3][3] | [3][2] | [3][1] | [3][0] >
Given :

int twoD[][] = new int [4][5];

برنامه بعدی هر عضو آرایه را از چپ به راست ، و از بالا به پایین شماره داده و سپس مقادیر آنها را نمایش می دهد :

```
// Demonstrate a two-dimensional array.
 class TwoDArray {
 public static void main(String args[] ){
 int twoD[][] = new int[4][5];
 int i, j,k = 0;
 for(i=0; i<4; i++)
 for(j=0; j<5; j++ ){
 twoD[i][j] = k;
 k++;
 }

 for(i=0; i<4; i++ ){
 for(j=0; j<5; j++)
 System.out.print(twoD[i][j] + " ");
```

```
System.out.println)(;
}
}
}
```

خروجی این برنامه بقرار زیر خواهد بود : 0 1 2 3 4

```
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

هنگام تخصیص حافظه به یک آرایه چند بعدی ، کافی است فقط حافظه برای اولین بعد را مشخص نمایید . می توانید ابعاد دیگر را جداگانه تخصیص دهید . بعنوان مثال ، کد زیر حافظه اولین بعد twoD را هنگام اعلان آن تخصیص می دهد . این کد حافظه دومین بعد را بصورت دستی اختصاص می دهد .

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

اگرچه در این حالت اختصاص انفرادی حافظه به دومین بعد هیچ مزیتی ندارد، اما احتمال چنین مزیتهایی وجود دارد . بعنوان مثال ، هنگامیکه ابعاد را بصورت دستی اختصاص می دهید ، نیازی نیست که همان ارقام برای اجزائ هر بعد را تخصیص دهید . همانطوریکه قبلا" گفتیم ، از آنجاییکه آرایه های چند بعدی واقعا" آرایه ای از آرایه ها هستند ، طول هر یک از آرایه ها تحت کنترل شما قرار می گیرند . بعنوان مثال ، برنامه بعدی یک آرایه دو بعدی ایجاد می کند که در آن اندازه های دومین بعد نامساوی هستند .

```
// Manually allocate differing size second dimension.
class TwoDAgain {
public static void main(String args[] ){



int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
```

```
twoD[3] = new int[4];

int i, j, k = 0;

for(i=0; i<4; i++)
for(j=0; j+ towD[i][j] = k; k++)
}

for(i=0; i<4; i++ ){
for(j=0; j+ System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

خروجی این برنامه بقرار زیر می باشد : 0

1 2

3 4 5

6 7 8 9

آرایه ای که توسط این برنامه ایجاد می شود ، بصورت زیر خواهد بود :

| [0][0] |

| [1][0] | [1][1] |

| [2][0] | [2][1] | [2][2] |

| [3][0] | [3][1] | [3][2] | [3][3] |

از آرایه های چند بعدی ناجور (یا نامنظم) در اکثر برنامه ها استفاده نمیشود زیرا برخلاف آنچه مردم هنگام مواجه شدن با یک آرایه چند بعدی انتظار دارند رفتار می کنند . اما این آرایه ها در برخی شرایط بسیار کارا هستند . بعنوان مثال ، اگر نیاز به یک آرایه دو بعدی خیلی بزرگ دارید که دارای تجمع پراکنده باشد (یعنی که یکی و نه همه اجزائ آن مورد استفاده قرار می گیرند) ، آنگاه آرایه بی قاعده احتمالا" یک راه حل کامل خواهد بود . این امکان وجود دارد که آرایه های چند بعدی را مقدار دهی اولیه نمود . برای

اینکار ، فقط کافی است هر یک از مقدار ده اولیه ابعاد را داخل مجموعه ابروهای ختص خودش قرار دهید . برنامه بعدی یک ماتریس ایجاد می کند که هر یک از اجزائ آن شامل حاصلضرب نمایه های سطرها و ستونها هستند. همچنین دقت نمایید که می توان از عبارات همچون مقادیر لفظی داخل مقدار ده اولیه آرایه استفاده نمود .

```
// Initialize a two-dimensional array.
class Matrix {
public static void main(String args[] ){
double m[][] = {
{ 0.0, 1.0, 2.0, 3.0 };
{ 0.1, 1.1, 2.1, 3.1 };
{ 0.2, 1.2, 2.2, 3.2 };
{ 0.3, 1.3, 2.3, 3.3 }};
int i, j;

for(i=0; i<4; i++ ){
for(j=0 j<4; j++)
System.out.print(m[i][j] + " ");
System.out.println)(;
}
}
}
```

پس از اجرای این برنامه ، خروجی آن بقرار زیر خواهد بود : 0 0 0 0

0 1 2 3
0 2 4 6
0 3 6 9

همانطوریکه مشاهده می کنید، هر سطر در آرایه همانگونه که در فهرستهای مقدار دهی اولیه مشخص شده ، مقدار دهی اولیه شده است . مثالهای بیشتری درباره استفاده از آرایه چند بعدی بررسی می کنیم . برنامه بعدی یک آرایه سه بعدی 3 x4x5ایجاد می کند . سپس حاصل نمایه های مربوطه را برای هر عضو بارگذاری می کند . در نهایت این حاصل ها را نمایش خواهد داد :

```
// Demonstrate a three-dimensional array.
class threeDDatrix {
```

```
public static void main(String args[] ){
int threeD[][][] = new int[3][4][5];
int i, j,k;
for(i=0; i<3; i++)
for(j=0; j<4; j++)
for(k=0; k<5; k++)
threeD[i][j][k] = i * j * k;

for(i=0; i<3; i++ ){
for(j=0; j<4; j++ ){
for(k=0; k<5; k++)
System.out.print(threeD[i][j][k] + " ");
System.out.println)(;
}
System.out.println)(;
}
}
}
```

خروجی این برنامه بقرار زیر خواهد بود :

```
0 0 0 0 0

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

دستور زبان جایگزین اعلان آرایه

یک شکل دوم برای اعلان یک آرایه بصورت زیر وجود دارد :

```
type [] var-name;
```

نام متغیر نوع

در اینجا کروشه ها بعداز مشخص کننده نوع می آیند نه بعداز نام متغیر آرایه . بعنوان مثال دو شکل اعلان زیر یکسان عمل می کنند :

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

دو شکل اعلان زیر هم یکسان عمل می کنند :

```
char twod1[][] = n
```

آرایه های دوباره ملاقات شده Arrays Revisited

آرایه ها بعنوان اشیائ پیاده سازی می شوند . بهمین دلیل ، یک خصلت ویژه وجود دارد که می توانید از مزیت آن استفاده نمایید . بطور اخص ، اندازه یک آرایه یعنی تعداد اعضایی که یک آرایه میتواند نگهداری نماید را می توان در متغیر نمونـه length پیـدا نمـود . کلیه آرایه ها این متغیر را دارند، و این متغیر همیشه اندازه آرایه را نگهداری می کند. در اینجا برنامه ای وجود دارد که ایـن خاصیت را نشان می دهد :

```
// This program demonstrates the length array member.
class Length {
public static void main(String args[] ){
int a1[] = new int [10];
int a2[] = {3, 5, 7, 1, 8, 99, 44,- 10};
int a3[] = {4, 3, 2, 1};
System.out.println("length of a1 is " + a1.length);
System.out.println("length of a2 is " + a2.length);
System.out.println("length of a3 is " + a3.length);
```

خروجی این برنامه بقرار زیر می باشد :

length of a1 is 10
length of a2 is 8
length of a3 is 4

همانطوریکه می بینید ، اندازه هر یك از آرایه ها بنمایش درآمده است . بیاد بسپارید که مقدار length کـاری بـا تعـداد اعضـایی کـه واقعا" مورد استفاده قرار گرفته اند، نخواهد داشت . این مقدار فقط منعکس کننده تعداد اعضـایی اسـت کـه آرایـه بـرای نگهـداری آن طراحی شده است .

در بسیاری از شرایط می توانید عضو length را در معرض کاربردهای مناسب قرار دهید . بعنوان مثـال ، یـک روایـت توسـعه یافتـه از کلاس stack را در اینجا مشاهده می کنید. احتمالا" بیاد دارید که روایتهای اولیـه ایـن کـلاس همیشـه یـک پشـته ده عضـوی ایجـاد میکرد . روایت بعدی همین برنامه ، به شما امکان ایجاد پشته هایی به هـر انـدازه دلخـواه را مـی دهـد . از مقـدار stck.length بـرای ممانعت از وقوع سرریزی پشته استفاده شده است .

```
// Improved Stack class that uses the length array member.
class Stack {
private int stck[];
private int tos;

// allocate and initialize stack
Stack(int size ){
stck = new int[size];
tos =- 1;
}

// Push an item onto the stack
void push(int item ){
if(tos==stck.length-1 )// use length member
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
```

```
// Pop an item from the stack
int pop )({
if(tos < 0 ){
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}

class TestStack2 {
public static void main(String args[] ){
Stack mystack1 = new Stack(5);
Stack mystack2 = new Stack(8);

// push some numbers onto the stack
for(int i=0; i<5; i++ )mystack1.push(i);
for(int i=0; i<5; i++ )mystack2.push(i);

// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<5; i++)
System.out.println(mystack1.pop))(;

System.out.println("Stack in mystack2:");
for(int i=0; i<8; i++)
System.out.println(mystack2.pop))(;
}
```

```
}
```

دقت کنید که برنامه فوق دو پشته ایجاد می کند : یکی با عمق پنج عضو ، و دیگری با عمق هشت عضو . همانطوریکه می بینید ، این حقیقت که آرایه ها اطلاعات length خودشان را نگهداری می کنند .

منابع :

http://www.irandevelopers.com/
http://docs.sun.com

نویسنده :
mamouri@ganjafzar.com محمد باقر معموری
ویراستار و نویسنده قسمت های تکمیلی :
zehs_sha@yahoo.com احسان شاه بختی

کتاب :

Java در 21 روز انتشارات نص
برنامه نویسی شی گرا انتشارات نص

نگاهی دقیق تر به روشها و کلاسها

از این پس موضوعات مربوط به روشها، شامل انباشتن (over loading) ، گذر دادن پارامترها (parameter passing)و خود فراخوانی یا همان برگشت پذیری (recursion) را بررسی می کنیم . و درباره کنترل دسترسی ، استفاده از واژه کلیدی Static و یکی از مهمترین کلاسهای توکار جاوا یعنی string بحث خواهیم نمود.

معرفی روشها

موضوع روشها بسیار گسترده است ، چون جاوا قدرت و انعطاف پذیری زیادی به روشها داده است . شکل عمومی یک روش بقرار زیر است :

```
type name(parameter-list ){
// body of method
}
```

در اینجا ، typeمشخص کننده نوع داده برگشت شده توسط روش است . این نوع هر گونه نوع معتبر شامل نوع کلاسی که ایجاد کرده اید ، می تواند باشد . اگر روش مقداری را برنمی گرداند ، نوع برگشتی آن باید Void باشد . نام روش توسط name مشخص می شود . این نام می تواند هر نوع شناسه مجاز باشد ، البته غیر از آنهایی که قبلا" برای اقلام داخل قلمرو جاری استفاده شده باشند parameter-list . یک پس آیند نوع و شناسه است که توسط یک کاما جدا می شود . پارامترها ضرورتا "متغیرهایی هستند که مقدار آرگومان (arguments) گذرکرده در روش ، هنگام فراخوانی را جذب می کنند . اگر روش فاقد پارامتر باشد ، آنگاه فهرست پارامتر تهی خواهد بود . روشهایی که بجای void یک نوع برگشتی (return) دارند ، مقداری را به روال فراخواننده با استفاده از شکل بعدی دستور returnباز می گردانند .

```
return value;
```

در اینجا value همان مقدار برگشتی است .

افزودن یک روش به کلاس Box

اگرچه خیلی خوب است که کلاسی ایجاد نماییم که فقط شامل داده باشد، اما بندرت چنین حالتی پیش می آید. در اغلب اوقات از روشها برای دسترسی به متغیرهای نمونه تعریف شده توسط کلاس ، استفاده می کنیم . در حقیقت ، روشها توصیف گر رابط به اکثر کلاسها هستند . این امر به پیاده سازان کلاس امکان می دهد تا صفحه بندی مشخصی از ساختارهای داده داخلی را در پشت تجریدهای (abstractions)

روشهای زیباتر پنهان سازند . علاوه بر تعریف روشهایی که دسترسی به داده ها را بوجود می آورند ، می توانید روشهایی را تعریف کنید که بصورت داخلی و توسط خود کلاس مورد استفاده قرار می گیرند . اجازه دهید با اضافه کردن یک روش به کلاس Box شروع کنیم . ممکن است درمثالهای قبلی شما هم احساس کرده باشید که محاسبه فضای اشغالی (Volume) یک box چیزی است که توسط کلاس Box در مقایسه با کلاس BoxDemo بسیار راحت تر مدیریت می شود . بعلاوه ، چون فضای اشغالی یک box بستگی به اندازه آن دارد، بنظر می رسد که بهتر است کلاس Box این محاسبه را انجام دهد . برای انجام اینکار، باید بصورت زیر یک روش را به Box اضافه نمایید :

```
// This program includes a method inside the box class.

class Box {
double width;
double height;
double depth;

// display volume of a box
void volume  (){
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}

class BoxDemo3 {
public static void main(String args[] ){
Box mybox1 = new Box ();
Box mybox2 = new Box ();

// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;

/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;

// display volume of first box
```

```
mybox1.volume ();

// display volume of second box
mybox2.volume ();
}
}
```

این برنامه خروجی زیر را تولید می کند که مطابق خروجی روایت قبلی همین برنامه است

```
Volume is 3000

Volume is 162
```

با دقت به دو خط بعدی کدها نگاه کنید :

```
mybox1.volume ();
mybox2.volume ();
```

خط اول در اینجا ، روش volume() را روی mybox1 قرار می دهد. یعنی که این خط volume() را بـا اسـتفاده از نـام شـی ئ کـه بـدنبال آن عملگـر نقطـه ای قـرار گرفتـه اسـت نسـبت بـه شـی ئ mybox1 فراخـوانی مـی کنـد . بـدین ترتیـب ، فراخـوانی mybox1.volume()فضای اشغالی box توصیف شده بوسیله mybox1 را نمـایش داده وفراخـوانی mybox2.volume()فضـای اشغالی box توصیف شده بوسیله mybox2 را نمایش می دهد. هر بار که volume() خواسته شود ، فضای اشغالی یک box مشـخص را نمایش خواهد داد . اگر با مفهوم فراخوانی یک روش نا آشنا هستید ، توصیف بعدی موضوع را تا حد زیادی روشن می کند . هنگامیکـه mybox1.volume()اجرا شود، سیستم حسن اجرای جاوا کنترل را به کدی که داخل volume() تعریف شده منتقل می کند . پس از اینکه دستور داخل volume() اجرا شود ، کنترل به روال فراخواننده برگشته و اجرای خط کد بعد از فراخوانی از سر گرفته خواهد شد . از یـک نقطـه نظـر عمـومی ، یـک روش (method) شـیوه جـاوا بـرای پیـاده سـازی زیـر روالهاسـت . (subroutions) یک نکته مهم درون روش volume() وجود دارد : ارجاع به متغیرهای نمونهwidth height و depth و بصورت مستقیم بـدون یـک نام شی ئ یا یک عملگر نقطه ای پیش آیند) قبلی (انجام می گیرد. وقتی یک روش از متغیر نمونه ای که توسط کـلاس خـود تعریـف شـده استفاده می کند ، اینکار را بطور مستقیم و بدون ارجاع صریح به یک شی ئ و بدون استفاده از عملگر نقطه ای انجام می دهد . اگر درباره آن بیندیشید ، بخوبی درک می کنید . یک روش همواره نسبت به شی ئ از کلاس خود قرار داده می شود . وقتی این تعبیه اتفاق می افتد ، شی ئ شناخته شده است . بدین ترتیب ، داخل یک روش نیازی به مشخص کردن یک شی ئ برای بار دوم وجود ندارد . این بـدان معنـی اسـت کـه width، height،و depth و داخل volume() بطور صریحی به کپیهای متغیرهای پیدا شده در آن شی ئ که volume() را فعال می کند ، ارجاع می کنند . اجازه دهید مجددا" مرور کنیم : وقتی یک متغیر نمونه بوسیله کدی که بخشی از کلاسـی کـه آن متغیـر نمونـه در آن

تعریف شده نیست مورد دسترسی قرار می گیرد اینکار باید از طریق یک شی ئ با استفاده از عملگر نقطه ای انجام شود. اما ، وقتی که متغیر نمونه بوسیله کدی که بخشی از همان کلاس مربوط به آن متغیر نمونه باشد مورد دسترسی قرار گیرد ، به آن متغیر می توان بطور مستقیم ارجاع نمود . همین مورد برای بررسی روشها نیز پیاده سازی می شود .

برگرداندن یک مقدار

در حالیکه پیاده سازی ()volume محاسبه فضای اشغالی یک box داخل کلاس مربوطهBox را حرکت می دهد ، اما بهترین راه نیست . بعنوان مثال ، اگر بخش دیگری از برنامه شما بخواهد فضای اشغالی یک box را بداند ، اما مقدار آن را نمایش ندهد چکار باید کرد ؟ یک راه بهتر برای پیاده سازی ()volume این است که آن را وادار کنیم تا فضای اشغالی box را محاسبه نموده و نتیجه را به فراخواننده (caller)برگرداند . مثال بعدی که روایت پیشرفته تر برنامه قبلی است ، همین کار را انجام می دهد :

```
// Now/ volume  ()returns the volume of a box.

class Box {
double width;
double height;
double depth;

// compute and return volume
doble volume  (){
return width * height * depth;
}
}

class BoxDemo4 {
public static void main(String args[] ){
Box mybox1 = new Box ();
Box mybox2 = new Box ();
double vol;

// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

```
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;

// get volume of first box
vol = mybox1.volume ();
System.out.println("Volume is " + vol);

// get volume of second box
vol = mybox2.volume ();
System.out.println("Volume is " + vol);
}
}
```

همانطوریکه می بینید ، وقتی volume() فراخوانی می شود ، در سمت راست یک دستور انتساب قرار می گیرد.در سمت چپ یک متغیر است که در این حالت vol میباشد که مقدار برگشتی توسط volume() را دریافت می کند . بدین ترتیب ، بعد از اجرای

```
vol = mybox.volume ();
```

مقدار mybox1.volume() برابر 3000 شده و سپس این مقدار در vol ذخیره خواهد شد . دو نکته مهم درباره برگرداندن مقادیر وجود دارد : ؤ نوع داده برگشت شده توسط یک روش باید با نوع برگشتی مشخص شده توسط همان روش سازگار باشد. بعنوان مثال ، اگر نوع برگشتی برخی روشها boolean باشد، نباید یک عدد صحیح را برگردان نمایید . ؤ متغیری که مقدار برگشتی توسط یک روش را دریافت می کند (نظیر vol در این مثال) باید همچنین با نوع برگشتی مشخص شده برای آن روش سازگاری داشته باشد . یک نکته دیگر : برنامه قبلی را می توان کمی کاراتر نیز نوشت زیرا هیچ نیاز واقعی برای متغیر vol وجود ندارد . فراخوانی volume() را می شد بطور مستقیم (بصورت زیر) در دستور println() استفاده نمود :

```
System.out.println("Volume is " + mybox1.volume90);
```

در این حالت ، هنگامیکه println() اجرا می شود ، mybox1.volume()بصورت خودکار فراخوانی شده و مقدار آن به println() گذر می کند .

افزودن روشی که پارامترها را می گیرد

اگرچه برخی روشها نیازی به پارامترها ندارند ، اما اکثر روشها این نیاز را دارند . پارامترها به یک روش ، امکان عمومی شدن را می دهند . یعنی که یک روش پارامتردار (parameterized) می تواند روی طیف گوناگونی از داده ها عمل کرده و یا در شرایط نسبتا" گوناگونی مورد استفاده قرار گیرد . برای مشاهده این نکته از یک برنامه خیلی ساده استفاده می کنیم . در اینجا روشی را مشاهده می کنید که مربع عدد 10 را برمی گرداند :

```
int square ()
{
return 10 * 10;
}
```

در حالیکه این روش در حقیقت مقدار مربع عدد 10 را برمی گرداند ، اما کاربرد آن بسیار محدود است . اما اگر روش را بگونه ای تغییر دهید که پارامتری را بگیرد همانطوریکه خواهید دید ، می توانید square() را مفیدتر بسازید .

```
int square(int i)
{
return i * i;
}
```

اکنون square() مربع هر مقداری را که فراخوانی شود ، برمی گرداند . یعنی که اکنون square() یک روش با هدف عمومی است که می تواند مربع هر مقدار عدد صحیح) بجای فقط عدد 10) را محاسبه می کند . مثالی را در زیر مشاهده می کنید :

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

در اولین فراخوانی square() مقدار 5 به پاارمتر i گذر می کند . در دومین فراخوانی ، iمقدار 9 را دریافت می کند . سومین فراخوانی مقدار y که در این مثال 2 است را می گذراند . همانطوریکه این مثالها نشان می دهند، square()قادر است مربع هر داده ای را که از آن می گذرد ، برگرداند . مهم است که دو اصطلاح پارامتر (parameter) و آرگومان (argument) را بوضوح درک نماییم . پارامتر ، متغیری است که توسط یک روش تعریف شده و هنگام فراخوانی روش ، مقداری را دریافت می کند .بعنوان مثال ، درsquare() ، iیک پارامتر است . آرگومان مقداری است که هنگام فراخوانی یک روش به آن گذر می کند . بعنوان مثال)´square 100(مقدار100 را بعنوان یک آرگومان گذر میدهد. داخلsquare() ، پارامترi مقداری را دریافت می کند . می توانید یک روش پارامتردار را برای توسعه

کلاس Box مورداستفاده قرار دهید. در مثالهای قبلی ، ابعاد هر یک box باید بصورت جداگانه با استفاده از یک سلسله دستورات نظیر مورد زیر ، تعیین می شدند :

```
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

اگرچه این کد کاری می کند، اما بدو دلیل دارای اشکال است . اول اینکه ، این کد بد ترکیب و مستعد خطا است . بعنوان مثال ، خیلی ساده امکان دارد تعیین یک بعد فراموش شود . دوم اینکه در برنامه های خوب طراحی شده جاوا ، متغیرهای نمونه فقط از طریق روشهای تعریف شده توسط کلاس خودشان قابل دسترسی هستند . در آینده قادر خواهید بود تا رفتار یک روش را تغییر دهید ، اما نمی توانید رفتار یک متغیر نمونه بی حفاظ (افشا شده) را تغییر دهید . بنابراین ، یک راه بهتر برای تعیین ابعاد یک box این است که یک روش ایجاد نماییم که ابعاد boxرا در پارامترهای خود نگهداشته و هر متغیر نمونه را بصورت مناسبی تعیین نماید . این برنامه توسط برنامه زیر ، پیاده سازی خواهد شد :

```
// This program uses a parameterized method.

class Box {
double width;
double height;
double depth;
// compute and return volume
double volume  (){
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d ){
width = w;
height = h;
depth = d;
}
}

class BoxDemo5 {
public static void main(String args[] ){
Box mybox1 = new Box ();
Box mybox2 = new Box ();
```

```
double vol;

// initialize each box
mybox.setDim(10, 20, 15);
mybox.setDim(3, 6, 9);

// get volume of first box
vol = mybox1.volume ();
System.out.println("Volume is " + vol);

// get volume of second box
vol = mybox2.volume ();
System.out.println("Volume is " + vol);
}
}
```

همانطوریکه می بینید ، روش setDim() برای تعیین ابعاد هر box مورد استفاده قرار می گیرد . بعنوان مثال وقتی

```
mybox1.setDim(10, 20, 15);
```

اجرا می شود ، عدد 10 به پارامتر w ، 20به h و 15به d کپی می شود . آنگاهheight وdepth و نسبت داده

نگاهی دقیق تر به گذر دادن آرگومانها

در کل ، دو راه وجود دارد تا یک زبان کامپیوتری یک آرگومان را به یک زیر روال گذر دهد. اولین راه " فراخوانی بوسیله مقدار-call)" by-value()است . این روش مقدار یک آرگومان را در پارامتر رسمی زیر روال کپی می کند . بنابراین تغییراتی که روی پارامتر زیر روال اعمال می شود ، تاثیری بر آرگومانی که برای فراخوانی آن استفاده شده نخواهد داشت . دومین راهی که یک آرگومان می تواند گذر کند " فراخوانی بوسیله ارجاع (call-by-reference()" است . در این روش ، ارجاع به یک آرگومان (نه مقدار آن آرگومان) به پارامتر گذر داده می شود . داخل زیر روال از این ارجاع برای دسترسی به آرگومان واقعی مشخص شده در فراخوانی استفاده می شود . این بدان معنی است که تغییرات اعمال شده روی پارامتر ، روی آرگومانی که برای فراخوانی زیر روال استفاده شده ، تاثیر خواهد داشت .خواهید دید که جاوا از هر دو روش برحسب اینکه چه چیزی گذر کرده باشد ، استفاده می کند . در جاوا ، وقتی یک نوع ساده را به یک روش گذر می دهید ، این نوع بوسیله مقدارش گذر میکند . بنابراین ، آنچه برای پارامتری که آرگومان را دریافت میکند اتفاق بیفتد هیچ تاثیری در خارج از روش نخواهد داشت . بعنوان مثال ، برنامه بعدی را در نظر بگیرید :

```
// Simple types are passed by value.
class Test {
void meth(int i, int j ){
i *= 2;
j /= 2;
}
}

class CallByValue {
public static void main(String args[] ){
Test ob = new Test ();
int a = 15, b = 20;
System.out.println("a and b before call :"  + a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call :" ++ a + " " + b);
}
}
```

خروجی برنامه فوق بقرار زیر می باشد :

```
a and b before call :15 20
a and b after call :15 20
```

بخوبی مشاهده می کنید که عملیات اتفاق افتاده داخل meth() هیچ تاثیری روی مقـادیر a و b و در فراخـوانی اسـتفاده شـده انـد، نخواهـد داشت . در اینجا مقادیر آنها به 30 و 10 تغییر نمی یابد . وقتی یک شی ئ را به یک روش گذر می دهید ، شـرایط بطـور مهیجـی تغییـر مـی کند زیرا اشیائ بوسیله ارجاعشان گذر داده می شوند . بیاد آورید که وقتی یک متغیر از یک نوع کلاس ایجاد می کنید ، شما فقط یک ارجاع به شی ئ خلق می کنید . بدین ترتیب ، وقتی این ارجاع را به یک روش گذر می دهید ، پارامتری که آن را دریافت مـی کنـد . بهمـان شـی ئ ارجاع می کند که توسط آرگومان به آن ارجاع شده بود . این بدان معنی است که اشیائ با استفاده از طریقه " فراخوانی بوسـیله ارجـاع " بـه روشها گذر داده می شوند. تغییرات اشیائ داخل روش سبب تغییر شیئی است که بعنوان یک آرگومان استفاده شده است . بعنوان مثال ، برنامـه بعدی را در نظر بگیرید :

```
// Objects are passed by reference.

class Test {
int a, b;

Test(int i, int j ){
a = i;
b = j;
}

// pass an object
void meth(Test o ){
o.a *= 2;
o.b /= 2;
}
}

class CallByRef {
public static void main(String args[] ){
Test ob = new Test(15, 20);

System.out.println("ob.a and ob.b before call :"  + ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call :" ++ ob.a + " " + ob.b);
```

برنامه فوق ، خروجی زیر را تولید می کند :

```
ob.a and ob.b before call :15 20
ob.a and ob.b after call :30 10
```

همانطوریکه می بینید ، در این حالت ، اعمال داخل meth() ، شیئی را که بعنوان یک آرگومان استفاده شده تحت تـاثیر قـرار داده اسـت . یک نکته جالب توجه اینکه وقتی یک ارجاع شی ئ به یک روش گذر داده می شود، خود ارجاع از طریق " فراخوانی بوسیله مقـدار " گـذر داده می شود . اما چون مقداری که باید گذر داده شود خودش به یک شی ئ ارجاع می کند ، کپی آن مقدار همچنان به همان شـی ئ ارجـاع می کند که آرگومان مربوطه ارجاع می کند . یاد آوری : وقتی یک نوع ساده به یک روش گذر داده میشود اینکار توسط " فراخوانی بوسـیله مقدار " انجام میگیرد .اشیائ توسط " فراخوانی بوسیله ارجاع " گذر داده می شوند

برگرداندن اشیائ

یک روش قادر است هر نوع داده شامل انواع کلاسی که ایجاد میکنید را برگرداند .بعنوان مثال ، در برنامه بعدی روش incrByTen() یک شی ئ را برمی گرداند که در آن مقدار a ده واحد بزرگتر از مقدار آن در شی ئ فراخواننده است .

```
// Returning an object.
class Test {
int a;

Test(int i ){
a = i;
}

Test incrByTen  (){
Test temp = new Test(a 10);
return temp;
}
}

class RetOb {
public static void main(String args[] ){
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen ();
System.out.println("ob1.a :" + ob1.a);
System.out.println("ob2.a :" + ob2.a);
ob2 = ob2.incrByTen ();
System.out.println("ob2.a after second increase :" + ob2.a);
}
```

```
}
```

خروجی برنامه فوق بقرار زیر می باشد :

```
ob1.a :2
ob2.a :12
ob2.a after second increase :22
```

همانطوریکه مشاهده می کنید ، هر بار که incrByTen() فراخوانده می شود ، یک شی ئ جدید تولید شده و یک ارجاع به آن شی ئ جدید به روال فراخواننده برگردان می شود . مثال قبلی یک نکته مهم دیگر را نشان میدهد : از آنجاییکه کلیه اشیائ بصورت پویا و با استفاده از new تخصیص می یابند ، نگرانی راجع به شیئی که خارج از قلمرو برود نخواهید داشت ، زیرا در این صورت روشی که شی ئ درآن ایجاد شده پایان خواهد گرفت . یک شی ئ مادامیکه از جایی در برنامه شما ارجاعی به آن وجود داشته باشد ، زنده خواهد ماند . وقتی که ارجاعی به آن شی ئ وجود نداشته باشد ، دفعه مرمت خواهد شد.

خود فراخوانی یا برگشت پذیری Recursion

جاوا از خود فراخوانی پشتیبانی می کند . خود فراخوانی پردازشی است که در آن چیزی بر حسب خودش تعریف شود . در ارتباط با برنامه نویسی جاوا ، خود فراخوانی خصلتی است که به یک روش امکان فراخوانی خودش را می دهد . روشی که خودش را فراخوانی می کند موسوم به " خود فراخوانده " یا برگشت پذیر (recursive) است . مثال کلاسیک برای خود فراخوانی محاسبه فاکتوریل یک رقم است . فاکتوریل یک عدد N عبارت است از حاصلضرب کلیه اعداد از 1تاN ا . بعنوان مثال فاکتوریل 3 معادل 1 x2x3یا عدد 6 است . در زیر نشان داده ایم چگونه می توان با استفاده از یک روش خود فراخوان ، فاکتوریل را محاسبه نمود :

```
// A simple example of recursion.
 class Factorial {
 // this is a recursive function
 int fact(int n ){
 int result;

 if(n==1 )return 1;
 result = face(n-1 )* n;
 return result;
 }
 }

 class Recursion {
 public static void mane(String args[] ){
 Factorial f = new Factorial ();

 System.out.println("Factorial of 3 is " + f.face(3));
 System.out.println("Factorial of 4 is " + f.face(4));
 System.out.println("Factorial of 5 is " + f.face(5));
 }
 }
```

خروجی این برنامه را در زیر نشان داده ایم :

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

اگر با روشهای خود فراخوان ناآشنا باشید ، آنگاه عملیات fact() ممکن است تا حدی بنظرتان گیج کننده باشد . طرز کار آن را توضیح داده ایم . وقتی fact() با یک آرگومان 1 فراخوانی می شود ، تابع مقدار 1 را برگردان می کند ، در غیر این صورت این تابع حاصلضرب

(n-1)fact *nرا برگردان میکند. برای ارزیابی این عبارت fact() را با n-1 فراخوانی می کنیم . این پردازش آنقدر تکرار می شود تا n مساوی 1باشد و فراخوانی های روش ، شروع به برگردان نمایند .

برای درک بهتر نحوه کار روش fact() اجازه دهید یک مثال کوتاه بیاوریم . وقتی فاکتوریل 3 را محاسبه می کنید، اولین فراخوانی fact()سبب دومین فراخوانی با آرگومان 2 می شود . این فراخوانی سبب می شود تا fact() برای سومین بار با آرگومان 1 فراخوانی شود . این فراخوانی مقدار 1 را برگردان می کند که بعدا" در 2) مقدار n در فراخوانی دوم) ضرب می شود. این نتیجه (یعنی عدد 2) آنگاه به فراخوانی اصلی fact() برگردان شده و در 3 (مقدار اصلی (n ضرب می شود . جواب کل عدد 6 است . اگر دستورات println() را در fact() جایگذاری نمایید بسیار جالب خواهد شد چون نشان خواهد داد که هر فراخوانی در چه سطحی است و جوابهای میانی چه مقادیری هستند . وقتی یک روش خودش را فراخوانی می کند ، حافظه پشته ها به متغیرهای محلی و پارامترهای جدید تخصیص داده می شود و کد روش نیز از همان اول با همین متغیرهای جدید اجرا می شود . یک خود فراخوانی ، کپی جدیدی از روش ایجاد نمی کند ، بلکه فقط آرگومانها جدید هستند . همچنانکه هر خود فراخوانی مقداری را برمی گرداند متغیرهای محلی و پارامترهای قدیمی از روی پشته برداشته می شوند ، و اجرا در نقطه ای از فراخوانی داخل روش از سر گرفته خواهد شد . روشهای خود فراخوان را می توان تلسکوپی نامید که باز و بسته می شوند . روایتهای خود فراخوانی بسیاری از روالها ، ممکن است کمی کندتر از روایتهای تکراری اجرا شوند و این بخاطر افزوده شدن انباشت فراخوانهای تابع اضافی است . بسیاری از خود فراخوانی ها به یک روش ممکن است سبب سر ریز شدن یک پشته شوند . از آنجاییکه ذخیره سازی پارامترها و متغیرهای محلی روی پشته انجام می گیرد و هر فراخوانی جدید یک کپی جدید از این متغیرها بوجود می آورد ، این امکان وجود دارد که پشته خراب شود . اگر چنین اتفاقی بیفتد ، سیستم حین اجرای جاوا یک استثنائ را بوجود می آورد. اما احتمالا" نگرانی درباره این مسائل نخواهید داشت مگر آنکه یک روال خود فراخوانی از حالت عادی خارج شود .

مهمترین مزیت روشهای خود فراخوان این است که از آنها برای ایجاد روایتهای ساده تر و روشنتر از الگوریتمهایی که می توان با رابطه های تکراری هم ایجاد نمود استفاده می شود . بعنوان مثال ، الگوریتم دسته بندی Quicksort در روش تکراری (iterative) بسیار بسختی پیاده سازی می شود . برخی مشکلات ، بخصوص مشکلات مربوط به Ai بنظر می رسد که نیازمند راه حلهای خود فراخوانی هستند. در نهایت ، اینکه بسیاری از مردم شیوه های خود فراخوانی را بهتر از شیوه های تکراری درک می کنند . هنگام نوشتن روشهای خود فراخوان ، باید یک دستور if داشته باشید که روش را مجبور کند تا بدون اجرای فراخوان خود فراخوان ، برگردان نماید . اگر اینکار را انجام ندهید ، هر بار که روش را فراخوانی کنید ، هرگز برگردان نخواهد کرد .

هنگام کار با خود فراخوانی ،این یکی از خطاهای رایج است .از دستورات println() هنگام توسعه برنامه بطور آزادانه استفاده کنید تا به شما نشان دهد چه چیزی در حال اتفاق افتادن است و اگر اشتباهی پیش آمده ، بتوانید اجرا را متوقف سازید .

دراینجا مثالی از خودفراخوانی را مشاهده کنید. روش خودفراخوانی printArray() اولین عنصر i در آرایه values را چاپ می کند :

```
// Another example that uses recursion.

class RecTest {
int values[];

RecTest(int i ){
values = new int[i];
}

// display array -- recursivaly
void printArray(int i ){
if(i==0 )return;
else printArray(i-1);
System.out.println("[" +( i-1 )+ "] " + values[i-1]);
}
}

class Recursion2 {
public static void mane(String args[] ){
RecTest ob = new RecTest(10);
int i;

for(i=0; i<10; i++ ) ob.values[i] = i;
ob.printArray(10);
}
}
```

این برنامه خروجی زیر را تولید می کند :

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
```

انباشتن روشها

در جاوا این امکان وجود دارد که دو یا چند روش را داخل یک کلاس که همان نام را دارد تعریف نمود ، البته مادامیکه اعلان پارامترهای آن روشها متفاوت باشد . در چنین شرایطی ، روشها را می گویند " انباشته شده " و این نوع پردازش را" انباشتن روش " می نامند . انباشتن روش یکی از راههایی است که جاوا بوسیله آن" چند شکلی " را پیاده سازی می کند . اگر تا بحال از زبانی که امکان انباشتن روشها را دارد استفاده نکرده اید ، این مفهوم در وهله اول بسیار عجیب بنظر می رسد . اما خواهید دید که انباشتن روش یکی از جنبه های هیجان انگیز و سودمند جاوا است . وقتی یک روش انباشته شده فراخوانی گردد ، جاوا از نوع و یا شماره آرگومانها بعنوان راهنمای تعیین روایت (Version) روش انباشته شده ای که واقعا" فراخوانی می شود ، استفاده می کند . بدین ترتیب ، روشهای انباشته شده باید در نوع و یا شماره پارامترهایشان متفاوت باشند . در حالیکه روشهای انباشته شده ممکن است انواع برگشتی متفاوتی داشته باشند ، اما نوع برگشتی بتنهایی برای تشخیص دو روایت از یک روش کافی نخواهد بود. وقتی جاوا با یک فراخوانی به یک روش انباشته شده مواجه می شود ، خیلی ساده روایتی از روش را اجرا می کند که پارامترهای آن با آرگومانهای استفاده شده در فراخوانی مطابقت داشته باشند . در اینجا یک مثال ساده وجود دارد که نشان دهنده انباشتن روش می باشد :

```
// Demonstrate method overloading.
class OverloadDemo {
void test  (){
System.out.println("No parameters");
}

// Overload test for one integer parameter.
void test(int a ){
System.out.println("a :" + a);
}

// Overload test for two integer parameters.
void test(int a, int b ){
System.out.println("a and b :" + a + " " + b);
}

// Overload test for a double parameter.
double test(double a ){
System.out.println("double a :" + a);
return a*a;
}
}
```

```
class Overload {
public static void main(String args[] ){
OverloadDemo ob = new OverloadDemo ();
double result;

// call all versions of test ()
ob.test ();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.2);
System.out.println("Result of ob.test(123.2 :)" + result);
}
}
```

این برنامه خروجی زیر را تولید می کند :

```
No parameters
a :10
a and b :10 20
double a :123.2
Result of ob.test(123.2 :)15178.2
```

همانطوریکه می بینید ، test()چهار بار انباشته شده است . اولین روایت پارامتری نمی گیرد ، دومین روایت یک پارامتر عدد صحیح می گیرد ، سومین روایت دو پارامتر عدد صحیح می گیرد و چهارمین روایت یک پارامتر double می گیرد . این حقیقت که چهارمین روایت test()همچنین مقداری را برمی گرداند، هرگز نتیجه حاصل از عمل انباشتن نیست ، چون انواع برگشتی نقشی در تجزیه و تحلیل انباشت ندارند .وقتی یک روش انباشته شده فراخوانی میشود،جاوا بدنبال تطبیقی بین آرگومانهای استفاده شده برای فراخوانی روش و پارامترهای آن روش می گردد . اما ، این تطابق نباید لزوما "همیشه صحیح باشد. در برخی شرایط تبدیل انواع خودکار جاوا میتواند نقشی در تجزیه و تحلیل انباشت داشته باشد . بعنوان مثال ، برنامه بعدی را در نظر بگیرید :

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
void test (){
System.out.println("No parameters");
}

// Overload test for two integer parameters.
void test(int a, int b ){
```

```java
System.out.println("a and b :" + a + " " + b);
}

// Overload test for a double parameter.
double test(double a ){
System.out.println("Inside test(double )a :" + a);
}
}

class Overload {
public static void main(String args[] ){
OverloadDemo ob = new OverloadDemo ();
int i = 88;

ob.test ();
ob.test(10, 20);

ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}
}
```

این برنامه خروجی زیر را تولید می کند: No parameters

```
a and b :10 20
Inside test(double )a :88
Inside test(double )a :123.2
```

همانطوریکه مشاهده میکنید، این روایت از overloadDemo تعریف کننده test(int) نمی باشد . بنابراین هنگامیکه test() همراه با یک آرگومان عدد صحیح داخل overload فراخوانی می شود ، هیچ روش تطبیق دهنده پیدا نخواهد شد . اما جاوا می تواند بطور خودکار یک عدد صحیح را به یک double تبدیل نماید و این تبدیل برای رفع فراخوانی مورداستفاده قرار میگیرد .بنابراین ، بعد از آنکه test(int) پیدا نمی شود ، جاوا i را به double ارتقائ داده و آنگاه test(double) را فراخوانی می کند . البته اگر test(int) تعریف شده بود ، فراخوانی می شد . جاوا فقط در صورتی که هیچ تطبیق دقیقی پیدا نکند ، از تبدیل خودکار انواع استفاده می کند . انباشتن روش از چند شکلی هم پشتیبانی می کند زیرا یکی از شیوه هایی است که جاوا توسط آن الگوی " یک رابط و چندین روش " را پیاده سازی می کند . برای درک این مطلب ، مورد بعدی را درنظر بگیرید.در زبانهایی که از انباشتن روش پشتیبانی

نمی کنند ، هر روش باید یک اسم منحصر بفرد داشته باشد . اما غالبا" می خواهید یک روش را بـرای چنـدین نـوع داده مختلـف پیـاده سـازی نمایید . مثلا" تابع قدر مطلق را در نظر بگیرید . در زبانهایی که از انباشتن روش پشتیبانی نمی کنند معمولا "سه یا چنـد روایـت مختلـف از ایـن تابع وجود دارد ، که هر یک اسم متفاوتی اختیار می کند . بعنوان نمونه ، در زبان C تابع abs() قدر مطلق یک عدد صحیح را برمی گرداند ، labs()قدر مطلق یک عدد صحیح long را برمی گرداند ، fabs()قدر مطلق یک عدد اعشاری را برمی گرداند . از آنجاییکه زبان C از انباشتن روش پشتیبانی نمی کند ، هر تابع باید اسم خاص خودش را داشته باشد ، حتی اگر هر سه تابع یک وظیفه واحد را انجام دهند. از نظر ذهنی این حالت ، شرایط پیچیده تری را نسبت به آنچه واقعا" وجود دارد ، ایجاب می کند . اگرچه مفهوم اصلی این توابع یکسـان اسـت ، امـا همچنان مجبورید سه اسم را بخاطر بسپارید . این شرایط در جاوا اتفاق نمی افتد ، زیرا تمامی روش های مربوط به قدر مطلق می توانند از یک اسم واحد استفاده نمایند . در حقیقت کتابخانه کلاس استاندارد جاوا (Java's standard) (class library شامل یک روش قدر مطلق موسوم به abs() می باشـد . ایـن روش توسـط کـلاس Math در جـاوا انباشـته شـده تـا کلیـه انـواع رقمـی را مـدیریت نمایـد . جـاوا بـر اساس نوع آرگومان ، تصمیم می گیرد که کدام روایت از abs() را فراخوانی نماید . ارزش انباشتن روشـها در ایـن اسـت کـه مـی تـوان بـا استفاده از یک اسم مشترک به کلیه روشهای مرتبط با هم دسترسی پیدا کرد. بدین ترتیب ،اسـم abs معـرف عمـل عمـومی اسـت کـه اجـرا خواهد شد . تعیین روایت مخصوص برای هر یک از شرایط خاص بر عهده کامپایلر می باشد . برنامه نویس فقط کافی است تـا اعمـال عمـومی که باید انجام شوند را بخاطر بسپارد . بدین ترتیب با استفاده از مفهوم چند شکلی ، چندین اسم به یک اسم خلاصه شده اند. اگرچـه ایـن مثـال بسیارساده بود، امااگر مفهوم زیربنایی آن را گسترش دهید ، می فهمید که انباشتن روشها تا چه حد در مدیریت پیچیدگی در برنامه ها سـودمند و کارساز است .

وقتی یک روش را انباشته می کنید ، هر یک از روایتهای آن روش قادرند هر نوع عمل مورد نظر شما را انجام دهند. هیچ قانونی مبنی بر اینکـه روشهای انباشته شده باید با یکدیگر مرتبط باشند، وجود ندارد. اما از نقطه نظر روش شناسی ، انباشتن روشها مستلزم یـک نـوع ارتبـاط اسـت . بدین ترتیب ، اگرچه می توانید از یک اسم مشترک برای انباشتن روشهای غیر مرتبط با هم استفاده نمایید ، ولی بهتـر اسـت ایـن کـار را انجـام ندهید . بعنوان مثال ، می توانید از اسم sqr برای ایجاد روشهایی که مربع یـک عـدد صـحیح و ریشـه دوم عـدد اعشـاری را برمـی گردانـد ، استفاده نمایید . اما این دو عمل کاملا" با یکدیگر متفاوتند. بکارگیری انباشتن روش درچنین مواقعی سبب از دست رفتن هدف اصلی این کار خواهد شد. در عمل ، فقط عملیات کاملا" نزدیک بهم را انباشته می کنید .

انباشتن سازندگان Overloading constructors

علاوه بر انباشتن روشهای معمولی ، می توان روشهای سازنده را نیز انباشته نمود. در حقیقت برای اکثر کلاسهایی که در دنیای واقعی ایجاد می کنید ، سازندگان انباشته شده بجای استثنائ یک عادت هستند . در زیر آخرین روایت Box را مشاهده می کنید :

```
class Box {
double width;
```

```
double height;
double depth;

// This is the constructor for Box.
Box(double w/ double h/ double d ){
width = w;
height = h;
depth = d;
}

// compute and return volume
double volume  (){
return width * height * depth;
}
}
```

همانطوریکه می بینید ، سازنده Box() نیازمند سه پارامتر است . یعنی کلیه اعلانات اشیائ Box باید سه آرگومان به سازنده Box() بگذرانند . بعنوان مثال دستور بعدی فعلا" نامعتبر است ;Box ob = new Box () :

ازآنجاییکه Box() نیازمند سه آرگومان است ، فراخوانی آن بدون آرگومانها خطا است . این مورد، سوالات مهمی را پیش رو قرار می دهد. اگر فقط یک box را بخواهید و اهمیتی نمی دهید و یا نمی دانید که ابعاد اولیه آن چه بوده اند ، چه پیش می آید ? همچنین ممکن است بخواهید یک مکعب را با مشخص کردن یک مقدار که برای کلیه ابعاد آن استفاده می شوند، مقدار دهی اولیه نمایید ? آنگونه که قبلا" کلاس Boxنوشته شده ، این گزینه ها در دسترس شما نخواهد بود . خوشبختانه راه حل این مشکلات کاملا" ساده است : خیلی ساده تابع سازنده Boxرا انباشته کنید بگونه ای که شرایط توصیف شده را اداره نماید . برنامه بعدی شامل یک روایت توسعه یافته Box است که اینکار را انجام می دهد :

```
/* Here/ Box defines three constructors to initialize
the dimensions of a box various ways.
*/
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d ){
```

```
width = w;
height = h;
depth = d;
}

// constructor used when no dimensions specified
Box  (){
width =- 1; // use- 1 to indicate
height =- 1; // an uninitialized
depth =- 1; // box
}

// constructor used when cube is created
Box(double len ){
width = height = deoth = len;
}

// compute and return volume
double volume  (){
return width * height * depth;
}
}

class OverloadDemo {
public static void main(String args[] ){
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box ();
Box mycube = new Box(7);

double vol;

// get volume of first box
vol = mybox1.volume ();
System.out.println("Volume of mybox1 is " + vol);

// get volume of second box
vol = mybox2.volume ();
```

```
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume ();
System.out.println("Volume of mycube is " + vol);
}
}
```

خروجی این برنامه بقرار زیر می باشد :

```
Volume of mybox1 is 3000
Volume of mybox2 is- 1
Volume of mycube is 343
```

neاجرا می شود ، پارامترها را مشخص

لغو (یا جلوگیری از پیشروی) روش

در یک سلسله مراتب کلاس ، وقتی یک روش در یک زیر کلاس همان نام و نوع یک روش موجود در کلاس بالای خود را داشته باشد، آنگاه میگویند آن روش در زیر کلاس ، روش موجود در کلاس بالا را لغو نموده (یا از پیشروی آن جلوگیری می نماید) . وقتی یک روش لغو شده از داخل یک زیر کلاس فراخوانی می شود ، همواره به روایتی از آن روش که توسط زیر کلاس تعریف شده ، ارجاع خواهد نمود و روایتی که کلاس بالا از همان روش تعریف نموده ، پنهان خواهد شد . مورد زیر را در نظر بگیرید :

```
// Method overriding.
class A {
int i, j;

A(int a, int b ){
i = a;
j = b;
}

// display i and j
void show  (){
System.out.println("i and j :" + i + " " + j);
}
}

class B extends A {
int k;

B(int a, int b, int c ){
super(a, b);
k = c;
}

// display k -- this overrides show  ()in A
void show  (){
System.out.println("k :" + k);
}
```

```
}

class Override {
public static void main(String args[] ){
B subOb = new B(1, 2, 3);

subOb.show (); // this calls show ()in B
}
}
```

حاصل تولید شده توسط این برنامه بقرار زیر می باشد :

```
k:3
```

وقتی show() روی یک شی ئ از نوع B فراخوانی می شود ، روایتی از show که داخلB تعریف شده مورد استفاده قرار میگیرد. یعنی که ، روایت show() داخلB ، روایت اعلان شده در A را لغو می کند . اگر می خواهید به روایت کلاس بالای یک تابع لغو شده دسترسی داشته باشید ، این کار را با استفاده از super انجام دهید . بعنوان مثال ، در این روایت از B روایت کلاس بالای show() داخل روایت مربوط به زیر کلاس فراخوانی خواهد شد . این امر به کلیه متغیرهای نمونه اجازه می دهد تا بنمایش درآیند .

```
class B extends A {
int k;

B(int a, int b, int c ){
super(a, b);
k = c;
}

void show (){
super.show (); // this calls A's show ()
System.out.println("k :" + k);
}
}
```

اگر این روایت از A را در برنامه قبلی جایگزین نمایید، خروجی زیر را مشاهده می کنید :

```
i and j :1 2
k:3
```

در اینجا ، super.show()روایت کلاس بالای show() را فراخوانی می کند . لغو روش فقط زمانی اتفاق می افتد که اسامی و نوع دو روش یکسان باشند . اگر چنین نباشد ، آنگاه دو روش خیلی ساده انباشته (overioaded) خواهند شد . بعنوان مثال ، این روایت اصلاح شده مثال قبلی را در نظر بگیرید :

```
// Methods with differing type signatures are overloaded -- not
// overridden.
class A {
int i, j;

A(int a, int b ){
i = a;
j = b;
}

// display i and j
void show  (){
System.out.println("i and j :" + i + " " + j);
}
}

// Create a subclass by extending class A.
class B extends A {
int k;

B(int a, int b, int c ){
super(a, b);
k = c;
}

// overload show ()
void show(String msg ){
System.out.println(msg + k);
}
}

class Override {
public static void main(String args[] ){
B subOb = new B(1, 2, 3);
```

```
subOb.show("This is k :"); // this calls show  ()in B
subOb.show (); // this calls show  ()in A
}
}
```

حاصل تولید شده توسط این برنامه بقرار زیر می باشد :

```
This is k:3
i and j :1 2
```

روایت show() درB ر یک پارامتر رشته ای می گیرد . این عمل سبب متفاوت شدن تاییدیه نوع آن از نـوع موجـود در A شـده ، کـه هـیچ پارامتری را نمی گیرد. بنابراین نباشتگی (یا مخفی شدن اسم) اتفاق نمی افتد .

روشهای Native

اگر چه بندرت اتفاق می افتد، اما گهگاه شرایطی پیش می آید که مـی خواهیـد یـک زیـر روال (subroutine) نوشـته شـده توسـط سـایر زبانهای غیر از جاوا را فراخوانی نمایید . معمولا" ، چنین زیر روالی بصورت کد قابل اجرا برای cpu و محیط خاصی که در آن کار می کنید عرضه می شود یعنی بصورت کد بومی . (Native) بعنوان مثال ممکن است بخواهید یک زیر روال کد بومی را فراخوانی کنید تـا بـه زمـان اجرای سریعتری برسید. یا ممکن است بخواهید از یک کتابخانه تخصصی شده متفرقه نظیر یک بسته آماری اسـتفاده نماییـد . امـا از آنجاییکـه برنامه های جاوا به کد بایتی کامپایل می شوند و سپس توسط سیستم حین اجرای جاوا تفسیر خواهند شد ، بنابراین بنظر می رسد فراخوانی یک زیر روال کد بومی از داخل برنامه جاوا غیر ممکن باشد .خوشبختانه ، این نتیجه گیری غلط است . جاوا واژه کلیدی native را تدارک دیده که برای اعلان روشهای کدهای بومی استفاده می شود. هربار که این روشها اعلان شوند می توانید آنها را از درون برنامه جاوای خود فراخوانی نمایید . درست مثل فراخوانی هر روش دیگری در جاوا . برای اعلان یک روش بومی ، اصلاحگر nativeرا قبـل از روش قـرار دهیـد. امـا بدنه ای برای روش تعریف نکنید ، بعنوان مثال :

```
public native int meth ();
```

هر بار که یک روش بومی را اعلان نمودید ، مجبورید روش بومی نوشته و یکسری مراحل پیچیده تر را تعقیب کنیـد تـا آن را بـه کـد جـاوای خود پیوند دهید . نکته : مراحل دقیقی که لازم است طی نمایید ممکن است برای محیط ها و روایتهای گوناگون جاوا متفاوت باشند . همچنین زبانی که برای پیاده سازی روش بومی اسـتفاده مـی کنیـد ، مـوثر خواهـد بـود . بحـث بعـدی از JDK ، (version 1.02)و ابزارهـای آن استفاده میکند. این یک محیط windows 95/NT را فرض میکند. زبانی که برای پیـاده سـازی روش بـومی اسـتفاده شـده ، زبـان C مـی باشد . آسانترین شیوه برای درک این پردازش ، انجام یک کار عملی است . برای شـروع برنامـه کوتـاه زیـر را کـه از یـک روش native موسوم به test() استفاده می کند وارد نمایید :

```
// A simple example that uses a native method.
public class NativeDemo {
int i;
int j;

public static void main(String args[] ){
NativeDemo ob = new NativeDemo ();

ob.i = 10;
ob.j = ob.test (); // call a native method
System.out.println("This is ob.j :" + ob.j);
}
```

```
// declare native method
public native int test ();

// load DLL that contains static method
static {
System.loadLibrary("NativeDemo");
}
}
```

دقت کنید که روش test() بعنوان native اعلان شده و بدنه ای نـدارد . ایـن روشـی اسـت کـه مـا در C باختصـار پیـاده سـازی میکنـیم . همچنین به بلوک static دقت نمایید. همانطوریکه قبلا" گفتیم : یک بلوک static فقط یکبار اجرا می شود ، و آنهـم زمـانی اسـت کـه برنامه شما شروع باجرا می نماید (یا دقیق تر بگویم ، وقتی که کلاس آن برای اولین بار بارگذاری می شـود) . در ایـن حالـت ، از ایـن بلـوک استفاده شده تا" کتابخانه پیوند پویا ()dynamic Link Library" را کـه دربرگیرنـده پیـاده سـازی بـومی test() اسـت ، بارگـذاری نماید . کتابخانه فوق توسط روش LoadLibrary() بارگذاری مسشود که بخشی از کلاس System است . شکل عمومی آن بقرار زیـر است :

Static void LoadLibrary(string filename)

در اینجا ، filenameرشته ای است که نام فایلی که کتابخانه را نگهداری میکند توصیف می کند . بـرای محـیط وینـدوز NT/95 فـرض شده که این فایل پسوند DLL ، داشته باشد . بعد از اینکه برنامه را وارد کردید، آن را کامپایل کنید تا NativeDemo.classتولید شـود. سپس ، باید ازjava.exeاستفاده نموده تا دو فایل تولید نمایید:

کـه NativeDemo.CوNativeDemo.1 و اسـت . شـما NativeDemo.h را در پیـاده سـازی test() مـی گنجانیـد . فایـل NativeDemo.Cیک فایل stub است که دربرگیرنده میزان کوچکی از کد است که رابط بین روش بومی شما و سیسـتم حـین اجـرای جاوا را تدارک می بیند. برای تولید NativeDemo.h ، از دستور بعدی استفاده نماییدjavahNativeDemo : این دستور یـک فایـل سرآیند موسوم به NativeDemo.h را ایجاد می کند. این فایل باید در فایـل C کـه test() را پیـاده سازی مـی کنـد ، گنجانـده شـود. حاصل تولید شده توسط این فرمان بقرار زیر می باشد :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include
/* Header for class NativeDemo */

#ifndef_ Included_NativeDemo
#define_ Included_NativeDemo
```

```
typedef struct ClassNativeDemo {
long j;
long j;
} ClassNativeDemo;
HandleTo(NativeDemo);

#ifdef __ cplusplus
extern "C" {
#endif
extern long NativeDemo_test(struct HNativeDemo *);
#ifdef __ cplusplus
}
#endif
#endif
```

به چند چیز مهم درباره این فایل دقت نمایید. اول اینکه ساختار clssNativeDemoدو عضو را دربرمی گیردi: و j. این ، متغیرهای نمونه تعریف شده توسط NativeDemo در فایل جاوا را معین می کند . دوم اینکه ، ماکرو HandleTo() ، نوع ساختار HNativeDemoرا ایجاد می کند ، که برای ارجاع به شیئی که test() را فراخوانی می کند مورد استفاده قرار می گیرد . به خط بعدی ، توجه ویژه ای مبذل دارید، که الگویی برای تابع test() که ایجاد کرده اید را تعریف می کند :

```
extern long NativeDemo_test(struct HNativeDemo *);
```

دقت نمایید که نام تابع ، NativeDemo-testاست . باید این را بعنوان نام تابع بومی که پیاده سازی می کنید ، بکار برید . یعنی که بجای ایجاد یک تابع C موسوم به test() ، یک تابع موسوم به NativeDemo-test() را ایجاد می کنید . پیشوند NativeDemo اضافه شده است چون مشخص می کند که روش test() بخشی از کلاس NativeDemoمی باشد . بیاد آورید ، کلاس دیگری ممکن است روش test() بومی خود را تعریف نماید که کاملا" با آنکه توسط NativeDemo اعلان شده ، متفاوت است . پیشوند گذاری نام روش بومی با نام کلاس فراهم کننده ، شیوه ای است برای تمایز بین روایتهای مختلف . بعنوان یک قانون عمومی ، توابع بومی ، یک نام کلاس که در آن اعلان شده اند را بعنوان یک پیشوند قبول می کنند . برگشت NativeDemo-test() از نوع long است ، اگرچه داخل برنامه جاوا که آن را فراخوانی می کند بعنوان یک int توصیف شده است . دلیل این امر بسیار ساده است . در جاوا ، اعداد صحیح مقادیر 32 بیتی هستند . در C ، یک عدد صحیح long لازم است حداقل 32 بیت برای یک نوع عدد صحیح تضمین نماید . یک چیز دیگر درباره الگوی NativeDemo-test() شایان توجه است . این الگو یک پارامتر از نوع HNativeDemo* تعریف می کند. کلیه روشهای بومی حداقل یک پارامتر دارند که اشاره گری است به شیئی که روش

بومی را فراخوانی نموده است . این پارامتر ضرورتا" شبیه This است . می توانید از این پارامتر استفاده کرده تا دسترسی بـه متغیرهـای نمونـه شیئی که روش را فراخوانی می کند ، داشته باشید

.برای تولید (NativeDemoC فایل stub) ، از این دستور استفاده کنید java-stubs NativeDemo :

فایل تولید شده توسط این دستور بصورت زیر می باشد :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include

/* Stubs for class NativeDemo */
/* SYMBOL :"NativeDemo/test ()I"/ Java_Native_test_stub */
__ declspec(dllexport )stack_item
*Java_NativeDemo_test_stub(stack_item *_P_/struct execenv *_ EE )_{
extern long NativeDemo_test(vide *);
_ P_[0].i = NativeDemo_test_(P_[0].p);
return_ P _+ 1;
}
```

شما این فایل را کامپایل نموده و با فایل پیاده سازی خود پیوند می دهید . پس از تولید فایلهای سرآیند و stub ضـروری ، مـی توانیـد پیـاده سازی خود از test() را بنویسید . از روایتی که در زیر نشان داده ایم ، استفاده نمایید :

```
/* This file contains the C version of the
test  ()method.
*/
#include
#include "NativeDemo.h"
#include
long NativeDemo_test(struct HNativeDemo *this)
{
printf("This is inside the native method.\n");
printf("this->i :%ld\n"/ unhand(this.-)i);
return( unhand(this-)>i);
}
```

دقت نمایید که این فایل دربرگیرنده stubpreamble.h است که شامل اطلاعات رابط سازی (interfacing) است . این فایل توسـط کامپایلر جاوا برای شما فراهم می شود . فایل سرآیند NativeDemo قبلا" توسط java ایجاد شده بود . پس از ایجاد test.c بایـد آن را و NativeDemo.C را کامپایل نمایید . بعد این دو فایل را بعنوان یک (DLL کتابخانه پیوند پویـا) بـا یکـدیگر پیونـد دهیـد . بـرای

انجام اینکار با کامپایلر میکروسافت C++/C ، از خط فرمان بعدی استفاده نمایید‌CL/LD NativeDemo.ctest.c : این فرمان یک فایل تحت عنوان NativeDemo.dll تولید می کند . هر بار که این کار انجام شود ، می توانید برنامه جاوا را اجرا کنید. انجام اینکار خروجی بعدی را تولید می کند :

```
This is inside the native method.
this->i :10
This is ob.j :20
```

داخل test.c به استفاده از unhand() توجه نمایید . این ماکرو توسط جاوا تعریف شده و یک اشاره گر را به نمونه ساختار class NativeDemo که در NativeDemo.h تعریف شده و همراهی شیئی است که test() را فراخوانی می کند ، برمی گرداند . در کل هرگاه نیاز به دسترسی به اعضائ یک کلاس جاوا داشته باشید ، از unhand() استفاده می کنید .

یادآوری : توضیحاتی که استفاده از native را محصور کرده اند، بستگی به پیاده سازی و محیط دارند . علاوه بر این ، رفتار مشخصی که در آن به کد جاوا رابط می سازید قابل تغییر است . باید مستندات موجود در سیستم توسعه جاوا خود را مطالعه نمایید تا جزئیات مربوط به روشهای بومی را درک کنید .

مشکلات مربوط به روشهای بومی

روشهای بومی بنظر می رسد تعهد بزرگی را پیشنهاد می کنند زیرا آنها به شما اجازه می دهند تا دسترسی به پایه موجود روالهای کتابخانه ای اتان را بدست آورده و امکان اجرای حین اجرای سریعتر را به شما عرضه می کنند . اما آنها همچنین دو مشکل اصلی را نشان می دهند: فرار امنیت بالقوه و کاهش قابلیت حمل . اجازه دهید باختصار این دو مورد را بررسی نماییم .

یک روش بومی ، یک ریسک امنیتی را مطرح می کند . از آنجاییکه یک روش بومی کد واقعی ماشین را اجرا می کند ، می تواند به هر بخش از رایانه میزبان دسترسی داشته باشد. یعنی که ، کد بومی منحصر به محیط اجرایی جاوا نیست . این کد امکان حمله ویروسی را هم می دهد. بهمین دلیل ریز برنامه ها نمی توانند از روشهای بومی استفاده نمایند. همچنین ، بارگذاری DLL ها می تواند محدود شود و بارگذاری آنها منوط به تصدیق مدیر امنیتی باشد .
دومین مشکل ، قابلیت حمل است . چون کد بومی داخل یک DLL گنجانده شده ، باید روی ماشینی که در حال اجرای برنامه جاوا است ، حاضر باشد. بعلاوه ، چون هر روش بومی بستگی به cpu و سیستم عامل دارد، هر DLL بطور وراثتی غیر قابل حمل میشود .بدین ترتیب ، یک برنامه جاوا که از روشهای بومی استفاده می کند، فقط قادر است وی یک ماشین که برای آن یک DLL سازگار نصب شده باشد ، اجرا شود.

منابع :

http://www.irandevelopers.com/
http://docs.sun.com

نویسنده :
mamouri@ganjafzar.com محمد باقر معموری
ویراستار و نویسنده قسمت های تکمیلی :
zehs_sha@yahoo.com احسان شاه بختی

کتاب :

Java در 21 روز انتشارات نص
برنامه نویسی شی گرا انتشارات نص